PCT/GB 00/01691

#2

The Patent Office

INVESTOR IN PEOPLE

GB 00/1691

EU

The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

PRIORITY DOCUMENT

SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

REC'D 25 MAY 2000

WIPO PCT

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

I also certify that the attached copy of the request for grant of a Patent (Form 1/77) bears an amendment, effected by this office, following a request by the applicant and agreed to by the Comptroller-General.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.

Signed M Cocks

Dated 25 April 2000

This Page Blank (uspto)

Patents Form 1/77

Patents Act 1977
(Rule 16)

Request for grant of a patent

(See the notes on the back of this form. You can also get an explanatory leaflet from the Patent Office to help you fill in this form.)

The Patent Office

1- 4 MAY 1999

Fee: £0

1/77

The Patent Office

Cardiff Road
Newport
Gwent NP9 1RH

1.	Your reference	40220/BJM
2.	Patent application number *(The Patent Office will fill in this part)*	9910280.8 05MAY99 E444665-1 D01631 F01/7700 0.00 - 9910280.8
3.	Full name, address and postcode of the or of each applicant *(underline all surnames)*	AT & T Laboratories-Cambridge Limited 24A Trumpington Street Cambridge CB2 1QA
	Patents ADP number *(if you know it)*	765307400l
	If the applicant is a corporate body, give the country/state of incorporation	England
4.	Title of the invention	Low Latency Network
5.	Full name, address and postcode in the United Kingdom to which all correspondence relating to this form and translation should be sent	~~Reddie & Grose~~ ~~16 Theobalds Road~~ ~~LONDON~~ ~~WC1X 8PL~~ Marks & Clerk 4220 Nash Court Oxford Business Park South OXFORD OX4 2RU
	Patents ADP number *(if you know it)*	91001
6.	If you are declaring priority from one or more earlier patent applications, give the country and the date of filing of the or of each of these earlier applications and *(if you know it)* the or each application number	Country
7.	If this application is divided or otherwise derived from an earlier UK application, give the number and the filing date of the earlier application	Number of earlier application — Date of filing *(day/month/year)*
8.	Is a statement of inventorship and of right to grant of a patent required in support of this request? *(Answer 'Yes' if:* *a) any applicant named in part 3 is not an inventor, or* *b) there is an inventor who is not named as an applicant, or* *c) any named applicant is a corporate body.* *See note (d))*	YES

Fm 51/77 - 18.4.00

9. Enter the number of sheets for any of the following items you are filing with this form. Do not count copies of the same document.

Continuation sheets of this form

Description	43
Claim(s)	26
Abstract	1
Drawing(s)	15

10. If you are also filing any of the following, state how many against each item.

Priority documents	-
Translations of priority documents	-
Statement of inventorship and right to grant of a patent *(Patents Form 7/77)*	1
Request for preliminary examination and search *(Patents Form 9/77)*	1
Request for substantive examination *(Patents Form 10/77)*	-
Any other documents *(please specify)*	

11. I/We request the grant of a patent on the basis of this application.

Signature: Reddie & Grose

Date: 4 May 1999

12. Name and daytime telephone number of person to contact in the United Kingdom

B J MATHER
0171-242 0901

Warning
After an application for a patent has been filed, the Comptroller of the Patent Office will consider whether publication or communication of the invention should be prohibited or restricted under Section 22 of the Patents Act 1977. You will be informed if it is necessary to prohibit or restrict your invention in this way. Furthermore, if you live in the United Kingdom, Section 23 of the Patents Act 1977 stops you from applying for a patent abroad without first getting written permission from the Patent Office unless an application has been filed at least 6 weeks beforehand in the United Kingdom for a patent for the same invention and either no direction prohibiting publication or communication has been given, or such direction has been revoked.

Notes

a) *If you need help to fill in this form or you have any questions, please contact the Patent Office on 0645 500505.*

b) *Write your answers in capital letters using black ink or you may type them.*

c) *If there is not enough space for all the relevant details on any part of this form, please continue on a separate sheet of paper and write "see continuation sheet" in the relevant part(s). Any continuation sheet should be attached to this form.*

d) *If you have answered 'Yes' Patents Form 7/77 will need to be filed.*

e) *Once you have filled in the form you must remember to sign and date it.*

f) *For details of the fee and ways to pay please contact the Patent Office.*

DUPLICATE

LOW LATENCY NETWORK

This invention, in its various aspects, relates to the field of asynchronous networking, and specifically to: a memory mapped network interface; a method of synchronising between a sending application, running on a first computer, and a receiving application, running on a second computer, the computers each having a memory mapped network interface; a communication protocol; and a computer network.

Due to a number of reasons, traditional networks, such as Gigabit Ethernet, ATM, etc., have not been able to deliver high bandwidth and low latency to applications that require them. A traditional network is shown in Fig. 1. To move data from computer 200 to another computer 201 over a network, the Central Processing Unit (CPU) 202 writes data from memory 204 through its system controller 206 to its Network Interface Card (NIC) 210. Alternatively, data may be transferred to the NIC 210 using Direct Memory Access (DMA) hardware 212 or 214. The NIC 210 takes the data and forms network packets 216, which contain enough information to allow them to be routed across the network 218 to computer system 201.

When a network packet arrives at the NIC 211, it must be demultiplexed to determine where the data needs to be placed. In traditional networks this must be done by the operating system. The incoming packet therefore generates an interrupt 207, which causes software, a device driver in operating system 209, to run. The device driver examines the header information of each incoming network packet 216 and

~~determines the correct location in memory 205, for data~~ contained within the network packet. The data is transferred into memory using the CPU 203 or DMA hardware (not shown). The driver may then request that operating system 209 reschedule any application process that is blocked waiting for this data to arrive. Thus there is a direct sequence from the arrival of incoming packets to the scheduling of the receiving application. These networks therefore provide implicit synchronisation between sending and receiving applications and are called synchronous networks.

It is difficult to achieve optimum performance using modern synchronous network hardware. One reason is that the number of interrupts that have to be processed increases as packets are transmitted at a higher rate. Each interrupt requires that the operating system is invoked and software is executed for each packet. Such overheads both increase latency and the data transfer size threshold at which the maximum network bandwidth is achieved.

These observations have led to the development of asynchronous networks. In asynchronous networks, the final memory location within the receiving computer for received data can be computed by the receiving NIC from the header information of a received network packet. This computation can be done without the aid of the operating system.

Hence, in asynchronous networks there is no need to generate a system interrupt on the arrival of incoming data packets. Asynchronous networks therefore have the potential of delivering high bandwidth and low latency; much greater than synchronous networks. The Virtual Interface Architecture (VIA) is emerging as a standard for asynchronous networking.

Memory-mapped networks are one example of asynchronous networks. An early computer network using memory mapping is described in US patent No. 4,393,443.

A memory-mapped network is shown in Fig. 2. Application 222 running on Computer 220 would like to communicate with application 223 running on Computer 221 using network 224. A portion of the application 222's memory address space is mapped using the computer 220's virtual memory system onto a memory aperture of the NIC 226 as shown by the application's page-tables 228 (these page-tables and their use is well known in the art). Likewise, a portion of application 223's memory address space is mapped using computer 221's virtual memory system onto a memory aperture of the NIC 229 using the application 223's page-tables 231. Software is usually required to create these mappings, but once they have been made, data transfer to and from a remote machine can be achieved using a CPU read or write instruction to a mapped virtual memory address.

If application 222 were to issue a number of processor write instructions to this part of its address space, the virtual memory and I/O controllers of computer 220 will ensure that these write instructions are captured by the memory aperture of the NIC 226. NIC 226, determines the address of the destination computer 221 and the address of the remote memory aperture 225 within that computer. Some combination of this address information can be regarded as the network address, which is the target of the write.

All the aperture mappings and network address translations are calculated at the time that the connection between the address spaces of computers 220 and 221 is made. The process

of address lookups and translations at each stage in the system can be carried out using hardware.

After receiving a write, NIC 226 creates network packets using its packetisation engine 230. These packets are forwarded to the destination computer 221. At the destination, the memory aperture addresses of the incoming packets are remapped by the packet handler onto physical memory locations 227. The destination NIC 229 then writes the incoming data to these physical memory locations 227. This physical memory has also been mapped at connection set-up time into the address space of application 223. Hence application 223 is able, using page-tables 231 and the virtual memory system, to access the data using processor read and write operations.

Commercial equipment for building memory-mapped networks is available from a number of vendors, including Dolphin Interconnect Solutions. Industry standards, such as Scalable Coherent Interface (SCI) (IEEE Standard 1596-1992), have been defined for building memory mapped networks, and implementations to the standards are currently available.

SCI is an example of an asynchronous network standard, which provides poor facilities for synchronisation at the time of data reception. A network using SCI is disclosed in US Patent No. 5,819,075. Figure 3 shows an example of an SCI-like network, where application 242 on computer 240 would like to communicate with application 243 on computer 241. Let us suppose that application 243 has blocked waiting for the data. Application 242 transmits data using the methods described above. After sending the data, application 242 must then construct a synchronisation packet in local

memory, and program the event generator 244, in NIC 246, to send the synchronisation packet 248, to the destination node.

On receiving synchronisation packet 248, the NIC 245 on computer 241, invokes its event handler 247, which generates an interrupt 249 allowing the operating system 248 to determine that application 243 is blocked and should be woken up. This is called out-of-band synchronisation since the synchronisation packet must be treated as a separate and distinct entity and not as part of the data stream. Out-of-band synchronisation greatly reduces the potential of memory-mapped networks to provide high bandwidth and low latency.

In other existing asynchronous networks, such as the newly emerging Virtual Interface Architecture (VIA) standard, some support is provided for synchronisation. A NIC will raise a hardware interrupt when some data has arrived. However, the interrupt does not identify the recipient of the data, instead only indicates that some data has arrived for some communicating end-point.

While delivery of data can be achieved solely by hardware, the software task of scheduling between a large number of applications, each handling received data, becomes difficult to achieve. Software, known as a device driver, is required to examine a large number of memory locations to determine which applications have received data. It must then notify such applications that data has been delivered to them. This might include a reschedule request to the operating system for the relevant applications.

~~The present invention, in its various aspects, is defined in~~ more detail in the appended claims to which reference should now be made.

A first aspect of the invention provides a method of synchronising between a sending application on a first computer and a receiving application on a second computer, each computer having a main memory, and at least one of the computers having an asynchronous network interface, comprising the steps of:

providing the asynchronous network interface with a set of rules for directing incoming data to memory locations in the main memory of the second computer;

storing in the network interface one or more triggering value(s), each triggering value representing a state of a data transfer between the applications;

receiving, at the network interface, a data stream being transferred between the applications;

comparing at least part of the data stream received with the stored triggering values;

if the compared part of the data stream matches any stored triggering value, indicating that the triggering value has been matched; and

storing the data received in the main memory of the second computer at one or more memory location(s) in accordance with the said rules.

Another aspect of the invention provides an An asynchronous network interface for use in a host computer having a main memory and connected to a network, the interface comprising:

means for storing a set of rules for directing incoming data to memory locations in the main memory of the host computer;

a memory for storing one or more triggering value(s), each value representing a state of a data transfer between two or more applications in the computer network;

a receiver for receiving a data stream being transferred between two or more applications in the computer network;

comparison means for comparing at least part of the data stream received by the network interface with the stored triggering values; and

a memory for storing information identifying any matched triggering values.

A further aspect of the invention provides a method of passing data between an application on a first computer and remote hardware within a second computer or on a passive backplane, the first computer having a main memory and an asynchronous network interface, the method comprising the steps of:

providing the asynchronous network interface with a set of rules for directing incoming data to memory or I/O location(s) of the remote hardware;

storing in the network interface one or more triggering value(s), each triggering value representing a state of a data transfer between the application and the hardware;

receiving, at the network interface, a data stream being transferred between the application and the hardware;

comparing at least part of the data stream received with the stored triggering value(s);

indicating that a triggering value has been matched, if any compared part of the data stream matches a triggering value;

~~storing data transmitted in memory or I/O location(s)~~ of the remote hardware in accordance with the said rules; and

storing the data received in the main memory of the computer at one or more memory location(s) in accordance with the said rules.

A further aspect of the invention provides a method of arranging data transfers from one or more applications on a computer, the computer having a main memory, an asynchronous network interface, and a Direct Memory Access (DMA) engine having a request queue address common to all the applications, comprising the steps of:

the application requesting the network interface to store a triggering value corresponding to a property of the data block to be transferred;

an application requesting the DMA engine to transfer a block of data;

the network interface storing a triggering value corresponding to a property of the data block to be transferred, along with an identification of the application which requested the DMA transfer;

the network interface monitoring the data stream being sent by the applications and comparing at least part of the data stream with the triggering value(s) stored in its memory; and

if any triggering value matches, indicating that that triggering value has matched.

A yet further aspect of the invention provides a method of transferring data from a sending application on a first computer to a receiving application on a second computer,

each computer having a main memory, and a memory mapped network interface, the method comprising the steps of:

creating a buffer in the main memory of the second computer for storing data being transferred as well as data identifying one or more pointer memory location(s);

storing at said pointer memory location(s) at least one write pointer and at least one read pointer for indicating those areas of the buffer available for writes and for reads;

in dependence on the values of the WRP(s) and RDP(s), the sender application writing to the buffer;

updating the value of the WDP(s), after a write has taken place, to update the indication of the areas of the buffer available for reads and writes;

in dependence on the values of WRP(s) and RDP(s), the receiver application reading from the buffer; and

updating the value of the RDP(s), after a read has taken place, to update the indication of the areas of the buffer available for reads and writes.

Another aspect of the invention provides a computer network comprising two computers, the first computer running a sending application and the second computer running a receiving application, each computer having a main memory and a memory mapped network interface, the main memory of the second computer having: a buffer for storing data being transferred between computers as well as data identifying one or more pointer memory location(s);

means for reading at least one write pointer (WRP) and at least one read pointer (RDP) stored at (a) pointer memory location(s), for indicating those areas of the buffer available for writes and those areas available for reads;

the network interface of the second computer comprising:

a memory mapping;

means for reading data from the buffer in accordance with the contents of the WRP(s) and RDP(s); and

means for updating the value of the RDP(s), after a read has taken place, to update the indication of the areas of the buffer available for reads and writes.

A further aspect of the invention provides a method of sending a request from a client application on a first computer to a server application on a second computer, and sending a response from the server application to the client application, both computers having a main memory and a memory mapped network interface, the method comprising the steps of:

(A) providing a buffer in the main memory of each computer;

(B) the client application, providing software stubs which produce a marshalled stream of data representing the request;

(C) the client application sending the marshalled stream of data to the server's buffer;

(D) the server application unmarshalling the stream of data by providing software stubs which convert the marshalled stream of data into a representation of the request in the server's main memory;

(E) the server application processing the request and generating a response;

(F)the server application providing software stubs which produce a marshalled stream of data representing the response;

(G) the server application sending the marshalled stream of data to the client's buffer; and

(H) the client application unmarshalling the received stream of data by providing software stubs which convert the received marshalled stream of data into a representation of the response in the client's main memory.

Another aspect of the invention provides a method of arranging data for transfer as a data burst over a computer network comprising the steps of: providing a header comprising the destination address of a certain data word in the data burst, and a signal at the beginning or end of the data burst for indicating the start or end of the burst, the destination addresses of other words in the data burst being inferrable from the address in the header.

A further aspect of the invention provides a method of processing a data burst received over a computer network comprising the steps of:

reading a reference address from the header of the data burst, and

calculating the addresses of each data word in the burst from the position of that data word in the burst in relation to the position of the data word to which the address in the header corresponds, and from the reference address read from the header.

Another aspect of the invention provides a method of interrupting transfer of a data burst over a computer network comprising the steps of:

halting transfer of a portion of the data burst which has not yet been transferred, thereby splitting the data

~~burst into two burst sections, one which is transferred, and~~ one waiting to be transferred.

A further aspect of the invention provides a method of restarting the transfer of a data burst, after the transfer of that data burst has been interrupted, the method comprising the steps of:

calculating a new reference address for the untransferred data burst section from the address contained in the header of the whole data burst, and from the position in the whole data burst of the first data word of the untransferred data burst section in relation to the position of the data word to which the address in the header corresponds;

providing a new header for the untransferred data burst section comprising the new reference address; and transmitting the new header along with the untransferred data burst section.

The first aspect of the present invention addresses the synchronisation problem for memory mapped network interfaces. The present invention uses a network interface, containing snooping hardware which can be programmed to contain triggering values comprising either addresses, address ranges, or other data which are to be matched. These data are termed 'Tripwires'. Once programmed, the interface monitors the data stream, including address data, passing through the interface for addresses and data which match the Tripwires which have been set. On a match, the snooping hardware can generate interrupts or increment event counters, or perform some other application specified action. This snooping hardware is preferably based upon Content Addressable Memory (CAM). References herein to the

"data stream" refer to the stream of data words being transferred and to the address data accompanying them.

The invention thus provides in-band synchronisation by using synchronisation primitives which are programmable by user level applications, while still delivering high bandwidth and low latency. The programming of the synchronisation primitives can be made by the sending and receiving applications independently of each other and no synchronisation information is required to traverse the network.

A number of different interfaces between the network interface and an application can be supported. These interfaces include VIA and the forthcoming Next Generation Input/Output (NGIO) standard. An interface can be chosen to best match an application's requirements, and changed as its requirements change. The network interface of the present invention can support a number of such interfaces simultaneously.

The Tripwire facility supports the monitoring of outgoing as well as incoming data streams. These Tripwires can be used to inform a sending application that its DMA send operations have completed or are about to complete.

Memory-Mapped network interfaces also have the potential to be used for communication between hardware entities. This is because memory mapped network interfaces are able to pass arbitrary memory bus cycles over the network. As shown in Fig. 4, it is possible to set up a memory aperture 254, in the NIC 252 of Computer 250, which is directly mapped via

~~NIC 259, onto an address region 257 of the I/O bus 253 of~~ passive backplane 251.

Using existing memory mapped interfaces, such as DEC Memory Channel or Dolphin SCI, an application running on Computer 250, which requires use of the hardware device 255, would require a (usually software) process to interface between itself and the Network Interface card (NIC) 252. This is because the NIC 252, would not appear at the hardware level in computer 250 as an instance of the remote hardware device 255, but instead as a network card which has a memory aperture 254 mapped onto the hardware device.

In a further aspect of the invention, we have appreciated that the interface of the present invention can be programmed to present the same hardware interface as the remote hardware device 255, and so appear at the hardware level in computer 250 to be an instance of the remote hardware device. If the network card 252 were an interface according to the present invention, so programmed, the remote hardware device 255 would appear as physically located within computer 250, in a manner transparent to all software. The hardware device 255, is able to be physically located both at the remote end of a dedicated link, or over a general network. The invention will support both general networking activity and remote hardware communication simultaneously on a single network card.

Another aspect of the invention relates to a link-level communication protocol which can be used to support cut-through routing and forwarding. There is no need for an entire packet to arrive at a NIC, or any other network entity supporting the communication protocol, before data

transmission can be started on an outgoing link. The invention also allows large bursts of data to be handled effectively without the need for a small physical network packet size such as that employed by an ATM network, it being possible to dynamically stop and restart a burst and regenerate all address information using hardware.

A preferred embodiment of the various aspects of the invention will now be described with reference to the drawings in which:

Figure 5 shows two or more computers connected by an embodiment of the present invention, using Network Interface Cards (NICs);
Figure 6 shows in detail the various functional blocks comprising the NICs of Figure 5;
Figure 7 shows the functional blocks of the NIC loyed within a Field Programmable Gate Array (FPGA);
Figures 8 and 8e shows the communication protocol used in one embodiment of the invention;
Figure 9 shows schematically hardware communication according to an embodiment of the invention;
Figure 10 shows schematically a circular buffer abstraction according to one embodiment of the invention;
Figure 11 shows schematically the system support for discrete message communication using circular buffers;
Figure 12 shows a client-server interaction according to an embodiment of the invention;
Figure 13 shows how the system of the present invention can support VIA;
Figure 14 shows outgoing stream synchronisation according to an embodiment of the present invention; and

Figure 15 shows a client-server interaction according to an embodiment of the invention using a hardware data source.

Referring to Figure 5, computers 1, 2 use the present invention to exchange data. A plurality of other computers such as 3, may participate in the data exchange if connected via optional network switch 4.

Each computer 1, 2 is composed of a microprocessor central processing unit 5,57, memory 6,60, local cache memory 7,57, and system controller 8,58. The system controller 8,58 interacts with its microprocessor 5,57 to allow the microprocessor to exchange data with devices attached to I/O bus 9. Attached to I/O bus 9,59 are standard peripherals, such as a video adapter 10. Also attached to I/O bus 9,59 is one or more network interfaces, in the form of NICS 11,56 which represent an embodiment of this invention. In computers 1, 2 the I/O bus is a standard PCI bus conforming to PCI Local Bus Specification, Rev. 2.1, although any other bus capable of supporting bus master operations can be used with suitable modification of System Controller peripherals, such as video card 10, and the interface to NIC 11,56.

Referring to Figure 6, each NIC comprises a memory 18, 19, 20 for storing triggering values, a receiver 15 for receiving a data stream, a comparator for comprising part of the data stream with the triggering values and a memory 42 for storing information which will identify matched triggering values. More specifically, in the preferred embodiment each NIC 56, 11 is composed of a PCI to Local Bus bridge 12, a control Field Programmable Gate Array (FPGA) 13, transmit (Tx) serialiser 14, fibre-optic transceiver 15, receive (Rx) de-serialiser 16, address multiplexer and latch

17, CAM array 18, 19, 20, boot ROMs 21 and 22, static RAM 23, FLASH ROM 24, and clock generator and buffer 25, 26. Figure 6 also shows examples of known chips which could be used for each component, for example boot ROM 21 could be an Altera EPC1 chip.

Referring to Figure 7, FPGA 13 is comprised of functional blocks 27-62. The working of the blocks will be explained by reference to typical data flows.

Operation of NIC 11 begins by computer 1 being started or reset. This operation causes the contents of boot ROM 21 to be loaded into FPGA 13 thereby programming the FPGA and, in turn, causing state machines 28, 37, 40, 43, 45, 46 and 47 to be reset.

Clock generator 25 begins running and provides a stable clock for the Tx serialiser 14. Clock buffer/divider 26 provides suitable clocks for the rest of the system. Serialiser 14 and de-serialiser 16 are reset and remain in a reset condition until communication with another node is established and a satisfactory receive clock is regenerated by de-serialiser 16.

PCI bridge 12 is also reset and loaded with the contents of boot ROM 22. Bridge 12 can convert (and re-convert at the target end) memory access cycles into I/O cycles and support legacy memory apertures, and as the rest of the NIC supports byte-enabled (byte-wide as well as word-wide) transfers, ROM 22 can be loaded with any PCI configuration space information, and can thus emulate any desired PCI card transparently to microprocessor 5.

Immediately after reset, FLASH control state machine 47 runs and executes a simple microcode sequence stored in FLASH memory 24. Typically this allows the configuration space of another card such as 69 in Figure 9 to be read, and additional information to be programmed into bridge 12. Programming of the FLASH memory is also handled by state machine 47 in conjunction with bridge 12.

Data transfer could in principle commence at this point, but arbiter 40 is barred from granting bus access to Master state machine 37 until a status bit has been set in one of the internal registers 49. This allows software to set up the Tripwires during the initialisation stage.

Writes from computer 1 to computer 2 take place in the following manner. Microprocessor 5 writes one or more words to an address location defined by system controller 8 to lie within NIC 11's address space. PCI to local bus bridge 12 captures these writes and turns them into local bus protocol (discussed elsewhere in this document). If the writes are within the portion of the address space determined to be within the local control aperture of the NIC by register decode 48, then the writes take place locally to the Content Addressable Memory appropriate register, (CAM), Static RAM (SRAM) or FLASH memory area. Otherwise target state machine 28 claims the cycles and forwards them to protocol encoder 29.

At the protocol encoder, byte-enable, parity data and control information are added first to an address and then to each word to be transferred in a burst, with a control bit marking the beginning of the burst and possibly also a control bit marking the end of the burst. The control bit

marking the beginning of the burst indicates that address data forming the header of the data burst comprises the first "data" word of the burst. Xon/Xoff-style management bits from block 31 are also added here. This protocol, specific to the serialiser 14 and de-serialiser 16 is also discussed elsewhere in this document.

Data is fed on from encoder 29 to output multiplexer 30, reducing the pin count for FPGA 13 and matching the bus width provided by serialiser 14. Serialiser 14 converts a 23-bit parallel data stream at 62MHz to a 1-bit data stream at approximately 1.5Gbit/s; this is converted to an optical signal by transceiver 15 and carried over a fibre-optic link to a corresponding transceiver 15 in NIC 56, part of computer 2. It should be noted that other physical layers and protocols are possible and do not limit the scope of the invention.

In NIC 56, the reconstructed digital signal is clock-recovered and de-serialised to 62MHz by block 16. Block 32 expands the recovered 23 bits to 46 bits, reversing the action of block 30. Protocol decoder 33 checks that the incoming words have suitable sequences of control bits. If so, it passes address/data streams into command FIFO 34. If the streams have errors, they are passed into error FIFO 35; master state machine 37 is stopped; and an interrupt is raised on microprocessor 57 by block 53. Software is then used to decipher the incoming stream until a correct sequence is found, whereupon state machine 37 is restarted. When a stream arrives at the head of FIFO 34, master state machine 37 requests access to local bus 55 from arbiter 40. When granted, it passes first the address, then the following data onto local bus 55. Bridge 12 reacts to this

address/data stream by requesting access to I/O bus 59 from system controller 58. When granted, it writes the required data into memory 60.

Reads of computer 2's memory 60 initiated by computer 1 take place in a similar manner. However, state machine 28 after sending the address word sends no other words, rather it waits for return data. Data is returned because master state machine 37 in NIC 56 reacts to the arrival of a read address by requesting a read of memory 60 via I/O bus 59 and corresponding local bus bridge 12. This data is returned as if it were write data flowing from NIC 56 to NIC 11, but without an initial address. Protocol decoder 33 reacts to this addressless data by routing it to read return FIFO 36, whereupon state machine 28 is released from its wait and the microprocessor 5's read cycle is allowed to complete. Should the address region be marked in NIC 56's bridge 12 as read-prefetchable, then a number of words are returned; if state machine 28 continues requesting data as if from a local bus burst read, then subsequent words are fulfilled directly from read return FIFO 36.

Should NIC 56 need to raise an interrupt on microprocessor 5, remote interrupt generator 54 causes state machine 28 to send a word from NIC 56 to a mailbox register in NIC 11's bridge 12. This will have been configured by software to raise an interrupt on microprocessor 5.

Inevitably, since the clocks 25 in NICs 11 and 56 will run at slightly different frequencies, there will be occasional overrun conditions. Where the command FIFO 34 exceeds a pre-programmed threshold value, an Xoff bit is sent to the corresponding protocol encoder 29. This bit causes the

encoder to request that the sending state machine 28 stops, if necessary in mid burst. Logic in bridge 12 takes care of restarting the data burst when the corresponding Xon is received some time later. This logic calculates a new reference address for the unsent part of the data burst, using the reference address in the header of the whole data burst, and from a count of the number of data words which are sent before the transfer is stopped. As, in this embodiment, successive data words in a burst have successively incrementing destination addresses, the destination address of the first data word in the unsent part of the data burst can easily be calculated.

It is also possible that data may be read out of FIFO 34 faster than it is written in. In the event of this happening, master state machine 37 uses pipeline delay 38 to anticipate the draining of FIFO 34 and to terminate the data burst on local bus 55. It then uses the CAM address latch/counter 41 to restart the burst when more data arrives in FIFO 34.

'Tripwires' are triggering values, such as addresses, address ranges or other data, that are programmed into the NIC to be matched. Preferably, the trigging values used as tripwires are addresses. To meet timing requirements during address match cycles (as data flows through the NIC), three CAM devices are pipelined to reduce the match cycle time from around 70 nanoseconds to less than 30 nanoseconds.

The programming of Tripwires takes place by microprocessor 5 writing to PCI bridge 12 via system controller 8 and I/O bus 9. For the purpose of writing the Tripwire data, CAM array 18, 19, 20 appears like conventional RAM to

microprocessor 5. For write cycles, this is done by CAM controller 43 generating suitable control signals to enable all three CAMs 18, 19, 20 for write access. Address latch 44 passes data to the CAMs unmodified. Address multiplexer 41 is arranged to pass local bus data out on the CAM address bus where it is latched at the moment addresses are valid on the local bus by latch 17. For read cycles, the process is similar, except that only CAM 18 is arranged to be enabled for read access, and address latch/counter 44 has its data flow direction reversed. So far as microprocessor 5 is concerned, it sees the expected data returned, since the memory arrays in CAMs 18, 19, 20 either contain the same data, or internal flags indicating that particular segments of the memory array have not yet been written and should not participate in match cycles.

Owing to the nature of the address/data bus being comprised of bursts of data, according to the preferred local protocol, the actual data stream cannot be used for monitoring address changes. A burst starts with the address of the first data word followed by an arbitrary number of data words. The address of the data words is implicit and increments from the start address. For normal inbound or outbound data transfer operations, address latch/counter 44 is loaded with the address of each new data burst, and incremented each time a valid data item is presented on internal local bus 55. CAM control state machine 43 is arranged to enable each CAM 18, 19, 20 in sequence for a compare operation as each new address is output by latch/counter 44. This sequential enabling of the CAMs combined with their latching properties permits the access time for a comparison operation to be reduced by a factor of three (there being three CAMs in this implementation, other

implementations being possible) from 70ns to less than 30ns. The CAM op-code for each comparison operation is output from one of the internal registers 49 via address multiplexers 41 and 17. The op-code is actually latched by address multiplexer 17 at the end of a read/write cycle, freeing the CAM address bus to return the index of matched Tripwires after comparison operations.

The Tripwire data (i.e. the addresses to be monitored) is written to sequential addresses in the CAM array. During the comparison operation (cycle), all valid Tripwires are compared in parallel with the address of the current data, be it inbound or outbound. During the operation, masking operations may be performed, depending on the type of CAM used, allowing certain bits of the address to be ignored during the comparison. In this way, a Tripwire may actually represent a range of addresses rather than one particular address.

When the CAM array signals a match found (i.e. a Tripwire has been hit), it returns the address of the Tripwire (its offset in the CAM array) via the CAM address bus to the tripwire FIFO 42. Two courses of action are then possible, depending on how internal registers 49 have been programmed.

One course of action is for state machine 45 to request that an interrupt be generated by management logic 53. In this case, an interrupt is received by microprocessor 5, and software is run which services the interrupt. Normally this would involve microprocessor 5 reading the Tripwire address from FIFO 42, matching the address with a device-driver table, signalling the appropriate process, marking it runnable and rescheduling.

~~An alternative course of action is for state machine 45 to~~ cause records to be read from SRAM 23 using state machine 46. A record comprises a number of data words; an address and two data words. These words are programmed by the software just before the Tripwire information is stored in the CAM. When a Tripwire match is made, the address in LATCH 44 is left shifted by two to form an address index for SRAM 23. The first word is then read by state machine 46 and placed on local bus 55 as an address in memory 6. A fetch-and-increment operation is then performed by state machine 45, using the second and third words of the SRAM record to first AND and then OR, or else INCREMENT the data referred to in memory 6. A bit in the first word read by the state machine will indicate which operation it should take. In the case of an INCREMENT, the first data word also indicates the amount to increment by.

These alternatives enable the implementation of such primitives as an event counter incremented on tripwire matches, or the setting of a system reschedule flag. This mechanism enables multiple applications to process data without the requirement for hardware interrupts to be generated after receipt of each network packet.

While in the case of the interrupt followed by a Tripwire FIFO read, the device driver is presented with a listof endpoints which require attention. This list improves system performance as the device driver is not required to scan a large number of memory locations looking for such endpoints.

Since the device driver is not required to know where the memory locations which have been used for synchronisation

are. It is also not required to have any knowledge or take part in the application level communication protocol. All communication protocol processing can be performed by the application and different applications are free to use differing protocols for their own purposes, and one device driver instance may support a number of such applications.

There is also a problem connected with programming a DMA engine that is addressed by an aspect of the invention. Conventional access to DMA engines is moderated either by a single system device driver, which requires (slow) context switches to access, or by virtualisation of the registers by system page fault, also requiring (multiple) context switches. The problem is that it is not safe for a user level application to directly modify the DMA engine registers or a linked list DMA queue, because this must be done atomically. In most systems, user applications cannot atomically update the DMA queue as they can be descheduled at any moment.

The invention addresses this problem by using hardware FIFO 50 to queue DMA requests from applications. Each application wanting to request DMA transfers sets up a descriptor, containing the start address and the length of the data to be transferred, in its local memory and posts the address of the descriptor to the DMA queue, whose address is common to all applications. This can be arranged by mapping a single page containing the physical address of the DMA queue as a write-only page into the address space of all user applications as they are initialised.

As soon as DMA work queue FIFO 50 is not empty, local bus 55 is not busy and the DMA engine in bridge 12 is also not

busy, Master/Target/DMA arbiter 40 grants DMA state machine 51 access to local bus 55. Using the address posted by the application in FIFO 50, state machine 51 then uses bridge 12 to read the descriptor in memory 6 into the descriptor block 52. State machine 51 then posts the start address and length information held in block 52 into the DMA engine in bridge 12.

When the DMA process is complete, bridge 12 notifies state machine 51 of the completion. The state machine then uses data from descriptor block 52 to write back a completion descriptor in memory 6. Optionally, an interrupt can also be raised on microprocessor 5, although a Tripwire may already have been crossed to provide this notification early in order to minimise the delay bringing the relevant application back onto microprocessor 5's run queue. This is shown later in this document.

Should queue 50 be full, then state machine 51 writes a failure code back into the completion field of the descriptor that the application has just attempted to place on the queue. Thus the application does not need to read the status of the NIC in order to safely post a DMA request. All applications can safely share the same hardware posting address, and no time-consuming virtualisation or system device driver process is necessary.

Should any operation take longer than a preset number of PCI cycles, timeout logic 61 is activated to terminate the current cycle and return an interrupt through block 53.

Another aspect of the invention relates to the protocol which is preferably used by the NIC. This protocol uses an

address and some additional bits in its header. This allows the transfer of variable length packets with simple routines for Segmentation and Reassembly (SAR) that are transparent to the sending or receiving codes. This is also done without the need to have an entire packet arrive before segmentation, reassembly or forwarding can occur, allowing the data to be put out on the ongoing link immediately. This enables data to traverse many links without significantly adding to the overall latency. The packets may be fragmented and coalesced on each link, for example between the NIC and a host I/O bus bridge, or between the NIC and another NIC. We term this cut-through routing and forwarding. In a network carrying a large number of streams, cut-through forwarding and routing enables small packets to pass through the network without any delays caused by large packets of other streams. While other network physical layers such as ATM also provide the ability to perform cut-through forwarding and routing, they do so at the cost of requiring all packets to be of a fixed small size.

Figure 8 shows an example of how this protocol has been implemented using the 23-bit data transfer capability of HP's GLINK chipset (serialiser 14 and de-serialiser 16). PCI to local bus bridge 12 provides a bus of 32 address/data bits, 4 parity bits and 4 byte-enable bits. It also provides an address valid signal (ADS) which signifies that a burst is beginning, and that the address is present on the address/data bus. The burst continues until a burst last signal (BLAST) is set active, signifying the end of a burst. It provides a read/write signal, and some other control signals that need not be transferred to a remote computer. Figure 8A shows how this protocol is used to transfer an n

data word burst 63. The data traffic closely mirrors that used on the PCI bus, but uses fewer signals.

The destination address always precedes each data burst. Therefore, the bursts can be of variable size, can be split or coalesced, by generating fresh address words, or by removing address words where applicable. In the preferred embodiment, sequential data words are destined for sequentially incrementing addresses. However, data words having sequentially decrementing addresses might also be used, or any other pattern of addresses may be used so long as it remains easy to calculate. So far as the endpoints are concerned, exactly the same data is transferred to exactly the same locations. The benefits are that packets can be of any size at all, reducing the overhead of sending an address; packets can be split (and addresses regenerated to continue) by network switches to provide quality of service, and receivers need not wait for a complete packet to arrive to begin decoding work.

Also, the destination address given in the header may be for the 'nth' data word in the burst, rather than for the first, although using the first data word address is preferred.

Figure 8b shows how the protocol of Figure 8a is transcribed onto the G-LINK physical layer. The first word in any packet contains an 18-bit network address. Each word of 63 is split into two words in 64; the lower 16 bits carry high and low addresses or data, corresponding to the address/data bus; the next 4 bits carry either byte enables or parity data. During the address phase, the byte enable field (only 2 bits of which are available, owing to the limitations of G-LINK) is used to carry a 2-bit code indicating read, write

or escape packet use. Escape packets are normally used to carry diagnostic or error information between nodes, or as a means of carrying the Xon/Xoff-style protocol when no other data is in transit. The G-LINK nCAV signal corresponds to the ADS signal of 63; nDAV is active throughout the rest of the burst and the combination of nDAV inactive and nCAV inactive signals the end of a burst, or nCAV active indicates the immediate beginning of another burst.

Figure 8c, shows a read data burst 65; this is the same as a write burst 64, except data bit 16 is set to 0. On the outbound request, the data field contains the network address for the read data to be returned to. When the data for a read returns 66, it travels like a write burst, but is signified by there only being one nCAV active (signifying the network address) along with the first word. An additional bit, denoted FLAG in Figure 8, is used to cary Xon/Xoff sttyle information when a burst is in progress. It is not necessary therefore to break up a burst in order to send an Escape packet containing the Xon/Xoff information. The FLAG bit also serves as an additional end of packet indicator.

In Figure 8c, 67,68 shows an escape packet; after the network address, this travels with 68 or without 67 a payload as defined by data bit 16 in the first word of the burst.

In a full networked implementation, an extra network address word may precede each of these packets. Other physical layer or network layer solutions are possible, without compromise to this patent application, including fibre

~~channel parts (using 8B/10B encoding) and conventional~~ networks such as ATM or even Ethernet. The physical layer only needs to provide some means of identifying data from non-data and the start of one burst from the end of a previous one.

A further aspect of the invention relates to the distribution of hardware around a network. One use of a network is to enable one computer to access a hardware device whose location is physically distant. As an example, consider the situation shown in Figure 9, where it is required to display the images viewed by the camera 70, (connected a frame-grabber card 69) on the monitor which is, in turn, connected to computer 72. The NIC 73 is programmed from Boot ROM 22 to present the same hardware interface as that of the frame-grabber card 69. Computer 72 can be running the standard application program as provided by a third party vendor which is unaware that system has been distributed over a network. All control reads and writes to the frame-grabber 69, are transparently forwarded by the NIC 73, and there is no requirement for an extra process to be placed in the data path to interface between the application running on CPU 74 and the NIC 73. Passive PCI I/O back-plane 71, requires simply a PCI bus clock and arbiter i.e., no processor, memory or cache. These functions can be implemented at very low cost.

The I/O buses are conformant to PCI Local Bus Specification 2.1. This PCI standard supports the concept of a bridge between two PCI buses. It is possible to program the NIC 73 to present the same hardware interface as a PCI bridge between Computer 72 and passive back-plane 71. Such programming would enable a plurality of hardware devices to

be connected to back-plane 71 and controlled by computer 72 without the requirement for additional interfacing software. Again, it should be clear that the invention will support both general networking activity and this remote hardware communication, simultaneously using a single network card.

A circular buffer abstraction will now be discussed as an example of the use of the NIC by an application. The circular buffer abstraction is designed for applications which require a producer/consumer software stream abstraction, with the properties of low latency and high bandwidth data transmission. It also has the properties of responsive flow control and low buffer space requirements. Fig. 10 shows a system comprising two software processes, applications 102 and 103, on different computers 100, 101. Application 102 is producing some data. Application 103 is awaiting the production of data and then consuming it. The circular buffer 107, is composed of a region of memory on Computer 101 which holds the data and two memory locations - RDP 106 and WRP 109. WRP 109 contains the pointer to the next byte of data to be written into the buffer, while RDP 106 contains the pointer to the last byte of data to be read from the buffer. When the circular buffer is empty, then WRP is equal to RDP + 1 modulo wrap-around of the buffer. Similarly, the buffer is full when WRP is equal to RDP - 1. There are also private values of WRP 108 and RDP 111 in the caches of computer 100 and computer 101 respectively. Each computer 100,101 may use the value of WRP and RDP held in its own local cache memory to compute how much data can be written to or read from the buffer at any point in time, without the requirement for communication over the network.

~~When the circular buffer 107 is created, the producer sets~~ up a Tripwire 110, which will match on a write to the RDP pointer 106, and the consumer sets up a Tripwire 113, which will match on a write to the WRP pointer 109.

If consumer application 103 attempts to read data from the circular buffer 107, it first checks to see if the circular buffer is empty. If so, application 103 must wait until the buffer is not empty, determined when WRP 109 has been seen to be incremented. During this waiting period, application 103 may either block, requesting an operating system reschedule, or poll the WRP 109 pointer.

If producer application 102 decides to write to the circular buffer 107, it may do so while the buffer is not full. After writing some data, application 102 updates its local cached value of WRP 108, and writes the updated value to the memory location 109, in computer 101. When the value of WRP 109, is updated, the Tripwire 113, will match as has been previously described.

If consumer application 103 is not running on CPU 118 when some data is written into the buffer and Tripwire 113 matches, NIC 115 will raise a hardware interrupt 114. This interrupt causes CPU 118 to run device driver software contained within operating system 118. The device driver will service the interrupt by reading the tripwire FIFO 42 on NIC 115 and determine from the value read, the system identifier for application 103. The device driver can then request that operating system 118, reschedule application 103. The device driver would then indicate that the tripwire 113 should not generate a hardware interrupt until

application 103 has been next descheduled and subsequently another Tripwire match has occurred.

Note that the system identifier for each running application is loaded into internal registers 49, each time the operating system reschedules. This enables the NIC to determine the currently running application, and so make the decision whether or not to raise a hardware interrupt for a particular application given a Tripwire match.

Hence, once consumer application 103 is again running on the processor further writes to the circular buffer 107, by application 102, may occur without triggering further hardware interrupts. Application 103 now reads data from the circular buffer 107. It can read data until the buffer becomes empty (detected by comparing the values of RDP and WRP 111,109). After reading, application 102 will update its local value of RDP 111 and finally writes the updated value of RDP to memory location 106 over the network.

If producer application 102 had been blocked on a full buffer, this update of RDP 106 would generate a Tripwire match 110, resulting in application 102, being unblocked and able to write more data into the buffer 107.

In normal operation, application 102 and application 103 could be operating on different parts of the circular buffer simultaneously without the need for mutual exclusion mechanisms or Tripwire.

The most important properties of the data structure are that the producer and the consumer are able to process data without hindrance from each other and that flow control is

~~explicit within the software abstraction.~~ Data is streamed through the system. The consumer can remove data from the buffer at the same time as the producer is adding more data. There is no danger of buffer over-run, since a producer will never transmit more data than can fit in the buffer.

The producer only ever increments WRP 108, 109 and reads RDP 106, and the consumer only ever increments RDP 106, 111, and reads WRP 109. Inconsistencies in the values of WRP and RDP seen by either the producer or consumer either cause the consumer to not process some valid data (when RDP 106 is inconsistent with 111), or the producer to not write some more data (when WRP 109 is inconsistent with 108), until the inconsistency has been resolved. Neither of these occurrences cause incorrect operation or performance degradation so long as they are transient.
It should also be noted that on most computer architectures, including the Alpha AXP and Intel Pentium ranges, computer 100 can store the value of the RDP 106 pointer in its processor cache, since the producer application 102 only reads the pointer 106. Any remote writes to the memory location of the RDP pointer 106 will automatically invalidate the copy in the cache causing the new value to be fetched from memory. This process is automatically carried out and managed by the system controller 8. In addition, since computer 101 keeps a private copy of the RDP pointer 111 in its own cache, there is no need for any remote reads of RDP pointer values during operation of the circular buffer. Similar observations can also be made for the WRP pointer 109 in the memory of computer 101 and the WRP pointer 108 in the cache of computer 100. This feature of the buffer abstraction ensures that high performance and low latency are maintained. Responsive application level flow-

control is possible because the cached pointer values can be exposed to the user-level applications 102, 103.

A further enhancement to the above arrangement can be used to provide support for applications which would like to exchange data in discrete units. As shown in Fig. 11, and in addition to the system described in Fig. 10. The system maintains a second circular buffer 127, of updated WRP 129 values corresponding to buffer 125. This second buffer 127 is used to indicate to a consumer how much data to consume in order that data be consumed in the same discrete units as it were produced. Note that circular buffer 125 contains the data to be exchanged between the applications 122 and 123.

The producer, application 122 writes data into buffer 125, updating the pointer WRP 129, as previously described. Once data has been placed in buffer 125, application 122 then writes the new value of the WRP 129 pointer into buffer 127. At the same time it also manipulates the pointer WRP 131. If either of these write operations does not complete then the application level write operation is blocked until some data is read by the consumer application 123. The Tripwire mechanism can be used as previously described, for either application to block on either a full or empty buffer pair.

The consumer application 123 is able to read from both buffers 125 and 127, in the process updating the RDP pointers 133, 135 in its local cache and RDP pointers 124, 126 over the network in the manner previously described. A data value read from buffer 127 indicates an amount of data, which had been written into buffer 125. This value may be used by application level or library software 123, to

consume data from buffer 125 in the same order and by the same discrete amounts as it were produced by application 122.

The NIC can also be used to directly support a low latency Request/Response style of communication, as seen in client/server environments such as Common Object Request Broker Architecture (CORBA) and Network File System (NFS) as well as transactional systems such as databases. Such an arrangement is shown in Fig.12, where application 142 on computer 140 acts as a client requesting service from application 143 on computer 141, which acts as a server. The applications interact via memory mappings using two circular buffers 144 and 145, one contained in the main memory of each computer. The circular buffers operate as previously described, and also can be configured to transfer data in discrete units as previously described.

Application 142, the client, writes a request 147 directly into the circular buffer 145, via the memory mapped connection(s), and waits for a reply by waiting on data to arrive in circular buffer 144. Most Request/Response systems use a process known as marshalling to construct the request and use an intermediate buffer in memory of the client application to do the marshalling. Likewise marshalling is used to construct a response, with an intermediate buffer being required in the memory of the server application. Using the present invention, marshalling can take place directly into the circular buffer 145 of the server as shown. No intermediate storage of the request is necessary at either the client or server computers 140, 141.

The server application 143 notices the request (possibly using the Tripwire mechanism) and is able to begin unmarshalling the request as soon as it starts to arrive in the buffer 145. It is possible that the server may have started to process the request 149 while the client is still marshalling and transmitting, thus reducing latency in the communication.

After processing the request, the server writes the reply 146 directly into buffer 144, unblocking application 142 (using the Tripwire mechanism), which then unmarshalls and processes the reply 148. Again, there is no need for intermediate storage, and unmarshalling by the client may be overlapped with marshalling and transmission by the server.

A further useful and novel property of a Request/Response system built using the present invention, is that data may be written into the buffer both from software running on a CPU, or any hardware device contained in the computer system. Fig. 15 shows a Request/Response system which is a file serving application. The client application 262 writes a request 267 for some data held on disks controlled by 271. The server application 263 reads 269 and decodes the request from its circular buffer 265 in the manner previously described. It then performs authentication and authorisation on the request according to the particular application.

If the request for data is accepted, the server application 263 uses a two-part approach to send its reply. Firstly, it writes, into the circular buffer 264, the software generated header part of the reply 266. The server application 263 then requests 273 that the disk controller 271 send the required data part of the reply 272 over the network to

circular buffer 264. This request to the disk controller takes the form of a DMA request, with the target address being an address on I/O bus 270 which has been mapped onto circular buffer 264. Note that the correct offset is applied to the address such that reply data 272 from the disk is placed immediately following the header data 266.

Before initiating the request 273, the server application 263 can ensure that sufficient space is available in the buffer 264 to accept the reply data. Further, it is not necessary for the server application 263 to await the completion request 273. It is possible for the client application 262 to have set a Tripwire 274 to match once the reply data 272 has been received into buffer 264. This match can be programmed to increment the WRP pointer associated with buffer 264, rather than requiring application 263 to increment the pointer as previously described. If a request fails, then the client application 262 level timeout mechanism would detect and retry the operation.

It is also possible for the client application 262 to arrange that reply data 272 be placed in some other data structure, (such as a kernel buffer-cache page), through manipulation of 169 and 167 as described later. This is useful when 264 is not the final destination of the rept data, so preventing a final memory copy operation by the client. Server application 263 would be unaware of this client side optimisation.

By use of this mechanism, the processing load on the server is reduced. The requirement for the server application to wait for completion of its disk requests is removed. The requirement for high bandwidth streams of reply data to pass

through the server's system controller, memory, cache or CPU is also removed.

As previously stated, the NIC of the present invention could be used to support the Virtual Interface Architecture (VIA) Standard. Fig. 13 shows two applications communicating using VIA. Application 152 sends data to application 153, by first writing the data to be sent into a region of its memory, shown as block 154. Application 152 then builds a transmit descriptor 156, which describes the location of block 154 and the action required by the NIC (in this case data transmission). This descriptor is then placed onto the TxQueue 158, which has been mapped into the user-level address-space of application 152. Application 152 then finally writes to the doorbell register 160 in the NIC 162 to notify the NIC that work has been placed on the TxQueue 158.

Once the doorbell register 160 has been written, the NIC 162 can determine, from the value written, the address in physical memory of the activated TxQueue 158. The NIC 152 reads and removes the descriptor 156 from the TxQueue 158, determines from the descriptor 156, the address of data block 154 and invokes a DMA 164 engine to transmit the data contained in block 154. When the data is transmitted 168, the NIC 162 places the descriptor 156 on a completion queue 166, which is also mapped into the address space of application 152, and optionally generates a hardware interrupt. The application 152 can determine when data has been successfully sent by examining queue 166.

When application 153 is to receive data, it builds a receive descriptor 157 describing where the incoming data should be

placed, in this case block 155. Application 153 then places descriptor 157 onto RxQueue 159, which is mapped into its user-level address-space. Application 153 then writes to the doorbell register 161 to indicate that its RXQueue 159 has been activated. It may choose to either poll its completion queue 163, waiting for data to arrive, or block until data has arrived and a hardware interrupt generated. The NIC 165 in computer 151 services the doorbell register 161 write by first removing the descriptor 157 from the RxQueue 159. The NIC 165 then locates the physical pages of memory corresponding to block 155 and described by the receive descriptor 157. The VIA standard allows these physical pages to have been previously locked by application 153 (preventing the virtual memory system moving or removing the pages from physical memory). However, the NIC is also capable of traversing the page-table structures held in physical memory and itself locking the pages.

The NIC 165 continues to service the doorbell register write and constructs a Translation Look-aside (TLB) entry 167 located in SRAM 23. When data arrives corresponding to a particular VIA endpoint, the incoming address matches an aperture 169 in the NIC, which has been marked as requiring a TLB translation. This translation is carried out by state machine 46 and determines the physical memory address of block 155.

The TLB translation, having been previously set up, occurs with little overhead and the data is written 175 to appropriate memory block 155. A Tripwire 171 will have been arranged (when the TLB 167 entry was constructed) to match when the address range corresponding to block 155 is written to. This Tripwire match causes the firmware 173 (implemented

in state machine 51) to place the receive descriptor 157 onto completion queue 163 to invalidate the TLB mapping 167 and optionally generate an interrupt. If the RxQueue 159 has been loaded with other receive descriptors, then the next descriptor is taken and loaded into the TLB as previously described. If application 153 is blocked waiting for data to arrive, the interrupt generated will result, (after a device driver has performed a search of all the completion queues in the system), in application 153 being re-scheduled. If there is no TLB mapping for the VIA Aperture addresses, or the mapping is invalid, an error is raised using an interrupt. If the NIC 165 is in the process of reloading the TLB 167 when new data arrives, then hardware flow control mechanism 31 is used to control the data until a path to the memory block in computer 151 has been completed.

As an optional extension to the VIA standard, the NIC could also respond to Tripwire match 171 by placing an index on Tripwire FIFO 42, which could enable the device driver to identify the active VIA endpoint without searching all completion queues in the system.

This method can be extended to provide support for I20 and the forthcoming Next Generation I/0 (NGIO) standard. Here, the transmit, receive and completion queues are located on the NIC rather than in the physical memory of the computer, as is currently the case for the VIA standard.

As mentioned previously, another aspect of this invention is its use in providing support for the outbound streaming of data through the NIC. This setup is described in Fig. 14. It shows a Direct Memory Access (DMA) engine 182 on the NIC 183, which has been programmed in the manner previously

described by a number of user-level applications 184. These applications have requested that the NIC 183 transfer their respective data blocks 181 through the NIC 183, local bus 189, fibre-optic transceiver 190 and onto network 200. After each application has placed its data transfer request onto the DMA request queue 185, it blocks, awaiting a re-schedule, initiated by device driver 187. It can be important that the system maintains fair access between a large number of such applications, especially under circumstances where an application requires a strict periodic access to the queue, such as an application generating a video stream.

Data transferred over the network by the DMA engine 182, traverses local bus 189, and is monitored by the Tripwire unit 186. This takes place in the same manner as for received data, (both transmitted and received data pass through the NIC using the same local bus 55).

Each application, when programming the DMA engine 182 to transmit a data block, also constructs a Tripwire which is set to match on an address in the data block. The address to match could indicate that all or a certain portion of the data has been transmitted. When this Tripwire fires and causes a hardware interrupt 188, the device driver 187 can quickly determine which application should be made runnable. By causing a system reschedule, the application can be run on the CPU at the appropriate moment to generate more DMA requests. Because the device driver can execute at the same time that the DMA engine is transferring data, this decision can be made in parallel to data transfer operations. Hence, by the time that a particular application's data transfer requests have been satisfied, the system can ensure that the

application be running on the CPU and able to generate more requests.

ABSTRACT

LOW LATENCY NETWORK

Asynchronous network interface and method of synchronisation between two applications on different computers is provided. The network interface contains snooping hardware which can be programmed to contain triggering values comprising either addresses, address ranges or other data which are to be matched. These data are termed "trip wires". Once programmed, the interface monitors the data stream, including address data, passing through the interface for addresses and data which match the trip wires which have been set. On a match, the snooping hardware can generate interrupts for increment event counters, or perform some other application-specified action. This snooping hardware is preferably based upon Content-Addressable Memory.

The invention thus provides in-band synchronisation by using synchronisation primitives which are programmable by user level applications, while still delivering high bandwidth and low latency. The programming of the synchronisation primitives can be made by the sending and receiving applications independently of each other and no synchronisation information is required to traverse the network.

CLAIMS

1. A method of synchronising between a sending application on a first computer and a receiving application on a second computer, each computer having a main memory, and at least one of the computers having an asynchronous network interface, comprising the steps of:

providing the asynchronous network interface with a set of rules for directing incoming data to memory locations in the main memory of the second computer;

storing in the network interface one or more triggering value(s), each triggering value representing a state of a data transfer between the applications;

receiving, at the network interface, a data stream being transferred between the applications;

comparing at least part of the data stream received with the stored triggering values;

if any compared part of the data stream matches any triggering value, indicating that the triggering value has been matched; and

storing the data received in the main memory of the second computer at one or more memory location(s) in accordance with the said rules.

2. A method according to claim 1, in which the step of providing the asynchronous network interface with a set of rules comprises the step of establishing a mapping between information contained within the incoming data stream and one or more memory location(s) of the main memory of the second computer.

3. A method according to claim 2, in which the asynchronous network interface is a memory mapped network

interface, and in which the step of providing the memory mapped network interface with a set of rules comprises the step of establishing a mapping between addresses contained within the incoming data stream and one or more memory location(s) of the main memory of the second computer.

4. A method according to any of claims 1 to 3, further comprising storing in the asynchronous network interface an action, corresponding to each triggering value, which is to be carried out, in the event that the triggering value is matched, to indicate that the triggering value has been matched.

5. A method according to any of claims 1 to 4, comprising the step of sending an interrupt when a triggering value matches.

6. A method according to any of claims 1 to 5, comprising the step of changing the value of a counter when a triggering value is matched.

7. A method according to any preceding claim in which the triggering value(s) comprise(s) address data, and the part of the data stream compared with the stored triggering value(s) comprises address data.

8. A method according to any preceding claim, wherein the step of storing a triggering value is initiated by an application on one of the computers writing a triggering value to a memory location in the local control aperture within the address space of the network interface.

9. A method according to any preceding claim comprising the steps of accessing the main memory of the sending application, and outputting data therefrom.

10. A method according to any preceding claim comprising the step of mapping each physical destination address of the data being sent to a virtual memory address on a sending computer.

11. A method according to any preceding claim, both computers having an asynchronous network interface, comprising the step of sending the data stream from the sending network interface to the receiving network interface.

12. A method according to claim 11 comprising the step of mapping each virtual address of the received data stream to a physical address memory location of the main memory of the receiving computer.

13. A method according to any preceding claim comprising the step of writing the transferred data to the main memory of the receiving computer.

14. A method according to any preceding claim, each computer having a network interface also having an I/O bus, the method comprising the step of providing the network interface with a local bus, and a bridge for interfacing between the local bus and the I/O bus of the computer.

15. A method according to claim 14, comprising the step of loading the bridge with predetermined configuration data.

16. An asynchronous network interface, for use in a host computer having a main memory and being connected to a network, the interface comprising:

means for storing a set of rules for directing incoming data to memory locations in the main memory of the host computer;

a memory for storing one or more triggering value(s), each value representing a state of a data transfer between two or more applications in the computer network;

a receiver for receiving a data stream being transferred between two or more applications in the computer network;

comparison means for comparing at least part of the data stream received by the network interface with the stored triggering values; and

a memory for storing information identifying any matched triggering values.

17. An asynchronous network interface according to claim 16, in which the set of rules comprises a memory mapping.

18. An asynchronous network interface according to claim 16 or 17, further comprising means for performing an action corresponding to a matched triggering value.

19. An asynchronous network interface according to claim 16, 17 or 18, further comprising a local bus.

20. An asynchronous network interface according to claim 19, the host computer having an I/O bus, the interface further comprising a bridge for interfacing between the I/O bus of the computer and the local bus of the network interface.

21. An asynchronous network interface according to any of claims 16 to 20, wherein the comparison means comprises a content-addressable memory.

22. An asynchronous network interface according to claim 21, wherein the comparison means comprises two or more content-addressable memories which are arranged so as to conduct a pipelined comparison of the data stream received by the network interface.

23. An asynchronous network interface according to any of claims 16 to 22, further comprising receive and transmit serialisers.

24. An asynchronous network interface according to any of claims 16 to 23, comprising a memory for storing configuration data for the bridge.

25. An asynchronous network comprising two or more computers each having an asynchronous network interface according to any of claims 16 to 24.

26. A method of passing data between an application on a first computer and remote hardware within a second computer or on a passive backplane, the first computer having a main memory and an asynchronous network interface, the method comprising the steps of:

providing the asynchronous network interface with a set of rules for directing incoming data to memory or I/O location(s) of the remote hardware;

storing in the network interface one or more triggering value(s), each triggering value representing a state of a data transfer between the application and the hardware;

receiving, at the network interface, a data stream being transferred between the application and the hardware;

comparing at least part of the data stream received with the stored triggering value(s);

indicating that a triggering value has been matched, if any compared part of the data stream matches a triggering value;

and, when a data stream is being passed from the first computer to the remote hardware, storing data received by the remote hardware in memory or I/O location(s) of the remote hardware in accordance with the said rules; and,

when a data stream is being transferred from the remote hardware to the first computer, storing the data received in the main memory of the first computer at one or more memory location(s) in accordance with the said rules.

27. A method according to claim 26, in which the step of providing the asynchronous network interface with a set of rules comprises the step of establishing a mapping between information contained within the incoming data stream and one or more memory or I/O location(s) of the receiving computer or hardware.

28. A method according to claim 27, in which the asynchronous network interface is a memory mapped network interface, and in which the step of providing the memory mapped network interface with a set of rules comprises the step of the first computer establishing a mapping, either locally or remotely, between addresses contained within the incoming data stream and one or more memory or I/O location(s) of the receiving computer or hardware.

29. A method according to any of claims 26 to 28, further comprising storing in the asynchronous network interface an action, corresponding to each triggering value, which is to be carried out, in the event that the triggering value is matched, to indicate that the triggering value has been matched.

30. A method according to any of claims 26 to 29, comprising the step of sending an interrupt when a triggering value matches.

31. A method according to any of claims 26 to 30, comprising the step of changing the value of a counter when a triggering value matches.

32. A method according to any of claims 26 to 31, in which the triggering value(s) comprise(s) address data, and the part of the data stream compared with the stored triggering value(s) comprises address data.

33. A method according to any of claims 26 to 32, wherein the step of storing a triggering value is initiated by an application on a computer writing a triggering value to a

memory location in the local control aperture within the address space of the network interface.

34. A method according to any of claims 26 to 33, comprising the steps of accessing the main memory of the application, and outputting data therefrom.

35. A method according to any of claims 26 to 34, comprising the step of mapping each physical destination address of the data being sent, to a virtual memory address on a computer.

36. A method according to any of claims 26 to 35, both computers having an asynchronous network interface, comprising the step of sending the data stream from the sending network interface to the receiving network interface.

37. A method according to any of claims 26 to 36, comprising the step of mapping each virtual address of the received data stream to a physical memory address or I/O location of the receiving computer or remote hardware.

38. A method according to any of claims 26 to 37, comprising the step of writing the transferred data to the main memory of the receiving computer.

39. A method according to any of claims 26 to 38, each computer or passive backplane having a network interface also having an I/O bus, the method comprising the step of providing each network interface with a local bus, and a bridge for interfacing between the local bus and the I/O

bus of the computer or passive backplane.

40. A method according to claim 39, comprising the step of loading the bridge with predetermined configuration data.

41. A method according to claim 40, in which the configuration data includes configuration data relating to the remote hardware.

42. A method according to any of claims 26 to 41, each computer and/or passive backplane having an I/O bus, the method further comprising the steps of:

loading the network interface of one of the computer(s) and/or of the passive backplane with data for configuring it to capture one or more predefined interrupt signal(s) on the I/O bus of that computer or passive backplane;

transferring a captured interrupt signal over the network to a network interface of another computer or passive backplane; and

loading the network interface of one of the computer(s) or of the passive backplane to assert one or more predefined interrupt signal(s) on the I/O bus of that computer or passive backplane, on receipt of the said transferred captured interrupt signal.

43. A method of arranging data transfers from one or more applications on a computer, the computer having a main memory, an asynchronous network interface, and a Direct Memory Access (DMA) engine having a request queue address common to all the applications, comprising the steps of:

the application requesting the network interface to store one or more triggering value(s) corresponding to a data block to be transferred;

an application requesting the DMA engine to transfer a block of data;

the network interface storing one or more triggering value(s) corresponding to the data block to be transferred, along with an identification of the application which requested the DMA transfer;

the network interface monitoring the data stream being sent by the applications and comparing at least part of the data stream with the triggering value(s) stored in its memory; and

if any triggering value matches, indicating that that triggering value has matched.

44. A method according to claim 43, in which the application requests a DMA transfer by setting up a descriptor indicating the transfer required, and sending this descriptor to the DMA request queue address.

45. A method according to claim 43 or 44, in which after requesting a data transfer and storage of a triggering value, the application blocks until it receives a reschedule.

46. A method according to claim 43, 44 or 45, in which when a triggering value matches, a reschedule is sent to the application which requested the storage of that triggering value.

47. A method according to any of claims 43 to 46, in which, if the request queue is full when an application

attempts to add a new request, the network interface indicates to that application that its requested transfer has failed.

48. A method according to any of claims 44 to 47, further comprising the steps of reading the first descriptor in the request queue and retrieving data from the main memory of the computer in accordance with the contents of the descriptor.

49. A method according to claim 48, further comprising the step of transmitting the data retrieved from the main memory in accordance with the content of the corresponding descriptor.

50. A method according to any of claims 43 to 49, further comprising the step of interrupting the transfer of a data block if the transfer is not completed after a predetermined length of time from the start of that transfer.

51. A method of transferring data from a sending application on a first computer to a receiving application on a second computer, each computer having a main memory, and a memory mapped network interface, the method comprising the steps of:

creating a buffer in the main memory of the second computer for storing data being transferred as well as data identifying one or more pointer memory location(s);

storing at said pointer memory location(s) at least one write pointer and at least one read pointer for indicating those areas of the buffer available for writes and for reads;

~~in dependence on the values of the WRP(s) and RDP(s),~~ the sender application writing to the buffer;

updating the value of the WRP(s), after a write has taken place, to update the indication of the area(s) of the buffer available for reads and the area(s) available for writes;

in dependence on the values of WRP(s) and RDP(s), the receiver application reading from the buffer; and

updating the value of the RDP(s), after a read has taken place, to update the indication of the area(s) of the buffer available for reads and the areas(s) available for writes.

52. A method according to claim 51, in which the step of updating the value of the WRP(s) includes the sending application sending the updated value of the WRP to the main memory of the second computer, via the network.

53. A method according to claim 51 or 52, in which the first computer comprises a processing means with a cache memory, comprising the step of the sending application storing the value of the updated WRP in the cache memory.

54. A method according to claim 51, 52 or 53, in which the step of updating the value of the RDP(s) includes the receiving application sending the updated value of the RDP to the main memory of the first computer, via the network.

55. A method according to claim 51, 52, 53 or 54, in which the second computer comprises a processing means with a cache memory, the method comprising the step of the receiving application storing the value of the updated RDP in its cache memory.

56. A method according to any of claims 51 to 55, comprising the steps of:

the network interface of the second computer storing triggering value(s) corresponding to the address(es) of one or more write pointer(s) (WRP(s));

the network interface of the second computer monitoring the data stream received from the first computer and comparing at least part of the data stream with the triggering value(s) stored in its memory; and

if any triggering value matches, indicating that that triggering value has matched.

57. A method according to claim 56, in which when a triggering value is matched by the receipt of the WRP write instruction, a receiver interrupt is generated.

58. A method according to any of claims 51 to 57, further comprising the steps of:

providing a second buffer in the main memory of the second computer for storing write pointer data;

storing one or more second-buffer write pointer(s) and second-buffer read pointer(s) indicating the areas of the second-buffer available for writes and reads;

when the sending application writes to the first-buffer and updates the write pointer(s) of the first-buffer, writing to said second-buffer, in accordance with the value of the write pointer(s) and read pointer(s) of the second-buffer, the updated value of the write pointer of the first-buffer; and

updating the value of the second-buffer write pointer(s) to update the indication of the area(s) of the second-buffer available for writes and the areas(s) available for reads.

~~59. A method according to claim 58, further comprising~~ the steps of:

reading a first-buffer write pointer value from the second buffer, in dependence on the contents of the second-buffer read pointer(s) and second-buffer write pointer(s), and

reading from the first buffer in dependence on the value of a first-buffer pointer and the write pointer value read from the second buffer.

60. A method according to any of claims 51 to 59, further comprising the steps of:

the network interface of the first computer storing triggering value(s) corresponding to address(es) of one or more RDP(s);

the network interface of the first computer monitoring the data stream received from the second computer and comparing at least part of the data stream with the triggering value(s) stored in its memory; and

if any triggering value matches, indicating that that triggering value has matched.

61. A method according to claim 60, in which when the network interface of the first computer matches a triggering value by the receipt of an RDP write instruction, a sender interrupt is generated.

62. A method according to any of claims 51 to 61, in which the sending application blocks if the values of the WRP(s) and RDP(s) indicate that the buffer is full.

63. A method according to claim 62, in which the sending application is unblocked on receipt of an interrupt.

64. A method according to any of claims 51 to 63, in which the receiving application blocks if the values of the WRP(s) and RDP(s) indicate that the buffer is empty.

65. A method according to claim 64, in which the receiving application is unblocked on receipt of an interrupt.

66. A method according to any of claims 51 to 65, in which a write pointer of a buffer points to the buffer address where the next byte of data should be written in that buffer.

67. A method according to any of claims 51 to 66, in which a read pointer of a buffer points to the buffer address of the first byte of data to be read from that buffer.

68. A method according to any of claims 51 to 67, in which when an application has written to the end of a buffer, it next writes to the start of the buffer, depending on the value of the WRP(s) and RDP(s) corresponding to that buffer.

69. A method according to any of claims 51 to 68, in which when an application has read to the end of a buffer, it next reads from the start of the buffer, depending on the value of the WRP(s) and RDP(s) corresponding to that buffer.

70. A method according to any of claims 51 to 70, in which the value of one or more WRPs and/or RDPs is updated when a triggering value is matched in a network interface.

71. A computer network comprising two computers, the first computer running a sending application and the second computer running a receiving application, each computer having a main memory and a memory mapped network interface, the main memory of the second computer having:

a buffer for storing data being transferred between computers as well as data identifying one or more pointer memory location(s);

means for reading at least one write pointer (WRP) and at least one read pointer (RDP) stored at (a) pointer memory location(s), for indicating the areas of the buffer available for writes and the area(s) available for reads;

the network interface of the second computer comprising:

a memory mapping;

means for reading data from the buffer in accordance with the contents of the WRP(s) and RDP(s); and

means for updating the value of the RDP(s) after a read has taken place, to update the indication of the area(s) of the buffer available for reads and the area(s) available for writes.

72. A computer network according to claim 71, the network interface of the first computer comprising:

a mapping memory; and

means for sending data to the buffer of the second computer.

73. A computer network according to claim 71 or 72, the main memory of the second computer storing the value of at least one WRP.

74. A computer network according to claim 71, 72 or 73, in which one or more pointer memory location(s) are in the main memory of the first computer.

75. A computer network according to any of claims 71 to 74, in which one or more pointer memory location(s) are located in the main memory of the second computer.

76. A computer network according to any of claims 71 to 75, in which the first computer comprises a processing means with a cache memory, with one or more WRP(s) and/or RDP(s) stored in that cache memory.

77. A computer network according to any of claims 71 to 76, in which the second computer has a processing means with a cache memory, with one or more WRP(s) and/or RDP(s) stored in that cache memory.

78. A computer network according to any of claims 71 to 77, in which the network interface of the first computer comprises:

means for writing data to the buffer in accordance with the values of at least one RDP and one WRP, using its memory mapping; and

means for updating the value of the WRP(s) to update the indication of the area(s) of the buffer available for reads and the area(s) available for writes.

79. A computer network according to any of claims 71 to 78, in which the main memory of the second computer comprises a second buffer; and the computer network also having:

~~means for reading one or more write pointer(s) and~~ one or more read pointer(s) of the second buffer indicating the areas of the second buffer available for writes and those available for reads;

means for updating the write pointer(s) of the first buffer, when an application running on one of the computers writes to the first buffer;

means for writing to said second buffer, in accordance with the value of the write pointer(s) and read pointer(s) of the second buffer, the updated value of the write pointer of the first buffer; and

means for updating the value of the second buffer's write pointer(s) to update the indication of the area(s) of the second buffer available for reads and the area(s) availale for writes.

80. A computer network according to claim 79, further comprising means for storing one or more write pointer(s) of the second buffer indicating the areas of the second buffer available for reads and the area(s) available for writes.

81. A computer network according to any of claims 51 to 80, in which the first and/or second buffer is a circular buffer.

82. A computer network according to claim 79, 80 or 81, in which the network interface of the second computer also comprises:

means for reading a first-buffer WRP value from the second buffer in accordance with the values of the second-buffer WRP(s) and RDP(s);

means for updating the RDP(s) of the second buffer to update the indication of the areas of the second buffer available for reads and writes;

means for reading from the first buffer in accordance with the contents of the first-buffer WRP value read from the second buffer, and a first-buffer RDP; and

means for updating the value of the RDP(s) of the first buffer to update the indication of the area(s) of the first buffer available for reads and writes when an application running on the second computer reads from the first buffer.

83. A computer network according to any of claims 51 to 82, the network interface of one or both computers also comprising:

a memory for storing triggering value(s), corresponding to one or more address(es) of WRP(s) and/or RDP(s);

means for monitoring a data stream being transferred between the two computers and for comparing at least part of the data stream being transferred with the stored triggering value(s); and

means for indicating that a triggering value has been matched, when the part of the data stream being compared matches a triggering value.

84. A computer network according to claim 83, in which the means for indicating that a triggering value has been matched comprises means for generating an interrupt.

85. A method of sending a request from a client application on a first computer to a server application on a second computer, and sending a response from the server

application to the client application, both computers having a main memory and a memory mapped network interface, the method comprising the steps of:

(A) providing a buffer in the main memory of each computer;

(B) the client application, providing software stubs which produce a marshalled stream of data representing the request;

(C) the client application sending the marshalled stream of data to the server's buffer;

(D) the server application unmarshalling the stream of data by providing software stubs which convert the marshalled stream of data into a representation of the request in the server's main memory;

(E) the server application processing the request and generating a response;

(F)the server application providing software stubs which produce a marshalled stream of data representing the response;

(G) the server application sending the marshalled stream of data to the client's buffer; and

(H) the client application unmarshalling the received stream of data by providing software stubs which convert the received marshalled stream of data into a representation of the response in the client's main memory.

86. A method according to claim 85 in which in step (c) and/or step (g) the stream of marshalled data is sent according to the method of any of claims 51 to 70.

87. A method according to claim 85 or 86, comprising the step of the client and server stubs sending the marshalled

streams of data directly over the network, using the memory mapped network interfaces.

88. A method according to claim 85, 86 or 87, in which the sending and/or marshalling of a response by the server application may take place at the same time as the client application is unmarshalling the response from its buffer.

89. A method according to any of claims 85 to 88, in which the sending and/or marshalling of a request by the client application may take place at the same time as the server application is unmarshalling the request from its buffer.

90. A method according to any of claims 85 to 89, in which the response generated by the server application comprises two or more parts;

the server application providing software stubs which convert at least a first part of the response into a marshalled stream of data;

the server application sending the marshalled data stream representing the first part of the response to the client's buffer;

one or more parts of the response being provided by a hardware device in the server computer in the form of a marshalled stream of data; and

the hardware device sending its marshalled stream of data to the client's buffer.

91. A method according to claim 90, in which one or more parts of the response generated by the server application is provided by another software application running on the

second computer in the form of a marshalled stream of data; and

the software application sending its marshalled stream of data to the client's buffer.

92. A method according to claim 90 or 91, in which each part of the response is sent to an appropriate part of the client's buffer such that when all parts of the response have been received in the buffer, the contents of the buffer comprise a marshalled data stream representing the whole response from the server application.

93. A method according to any of claims 85 to 92, comprising the steps of:

the network interface of the first computer storing triggering value(s) corresponding to a property of one or more parts of the expected response;

the network interface of the first computer monitoring the response received from the server application and comparing at least part of the data stream with the triggering value(s) stored in its memory; and

if any triggering value matches, indicating that that triggering value has matched.

94. A method according to claim 93, comprising the step of sending an interrupt when a triggering value matches.

95. A method according to claim 93 or 94, comprising the step of changing the value of a counter when a triggering value is matched.

96. A method according to claims 93, 94 or 95, in which the client application, while it is waiting for the

response from the server application, blocks or polls an event counter.

97. A method of arranging data for transfer as a data burst over a computer network comprising the steps of: providing a header comprising the destination address of a certain data word in the data burst, and a signal at the beginning or end of the data burst for indicating the start or end of the burst, the destination addresses of other words in the data burst being inferrable from the address in the header.

98. A method according to claim 52, in which the signal identifying the end of a burst comprises a null signal.

99. A method of processing a data burst received over a computer network comprising the steps of:

reading a reference address from the header of the data burst, and

calculating the addresses of each data word in the burst from the position of that data word in the burst in relation to the position of the data word to which the address in the header corresponds, and from the reference address read from the header.

100. A method of interrupting transfer of a data burst over a computer network comprising the steps of:

halting transfer of a portion of the data burst which has not yet been transferred, thereby splitting the data burst into two burst sections, one which is transferred, and one waiting to be transferred.

101. A method of restarting transfer of a data burst that has been interrupted according to the method of claim 100, comprising the steps of:

calculating a new reference address for the untransferred data burst section from the address contained in the header of the whole data burst, and from the position in the whole data burst of the first data word of the untransferred data burst section in relation to the position of the data word to which the address in the header corresponds;

providing a new header for the untransferred data burst section comprising the new reference address; and transmitting the new header along with the untransferred data burst section.

102. A method according to claim 101, comprising calculating the new reference address for the untransferred data burst section from the reference address contained in the header of the whole data burst and from the number of data words in the transferred data burst section.

103. A memory mapped network interface substantially as described herein with reference to Figures 5 to 15.

104. A computer network comprising two or more computers having a memory mapped network interface substantially as described herein with reference to Figures 5 to 15.

105. A method of synchronising between a sending application on a first computer and a receiving application on a second computer, both computers having a

memory mapped network interface, substantially as described herein with reference to Figures 5 to 15.

106. A protocol substantially as described herein with reference to Figure 8.

107. A method of arranging data for transfer substantially as described herein with reference to Figures 5 to 15.

108. A method of processing data substantially as described herein with reference to Figures 5 to 15.

109. A method of interrupting transfer of a data burst substantially as described herein with reference to Figures 5 to 15.

110. A method of restarting transfer of a data burst substantially as herein described with reference to Figures 5 to 15.

111. A method of arranging data transfers from one or more applications on a computer, the computer having a main memory, a memory mapped network interface, and a Direct Memory Access (DMA) engine having a request queue address common to all the applications, the method being substantially as described herein.

112. A method of transferring data from a first application on a first computer to a second application on a second computer, the method being substantially as described herein with reference to Figure 10.

113. A method of passing data between an application on a first computer and remote hardware within a second computer or on a passive backplane, the method being substantially as described herein.

114. A method of sending a request from a client application to server application and sending a response from the server application to the client application, the method being substantially as herein described herein.

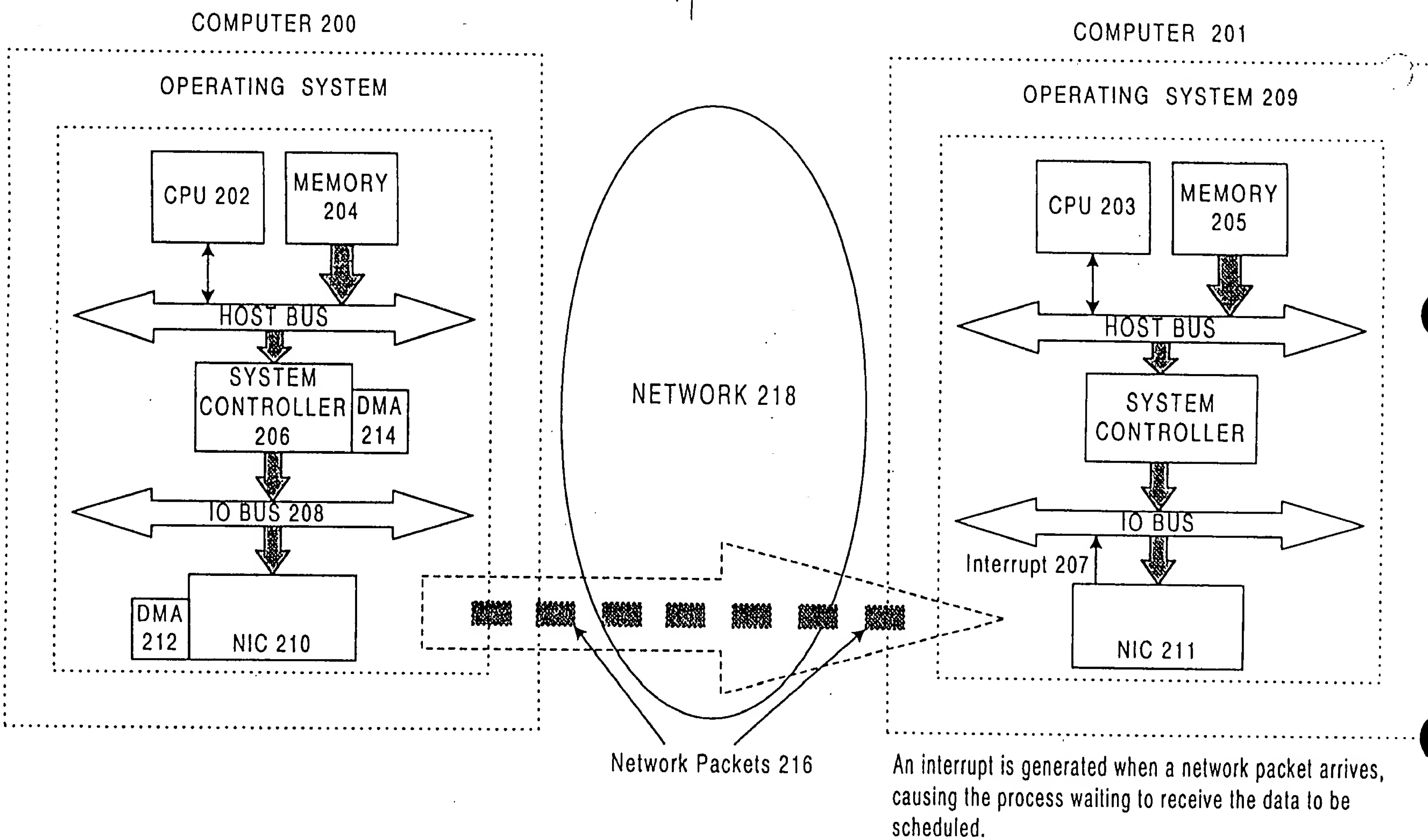


Fig 1. Prior Art. Synchronous Computer Networks

This Page Blank (uspto)

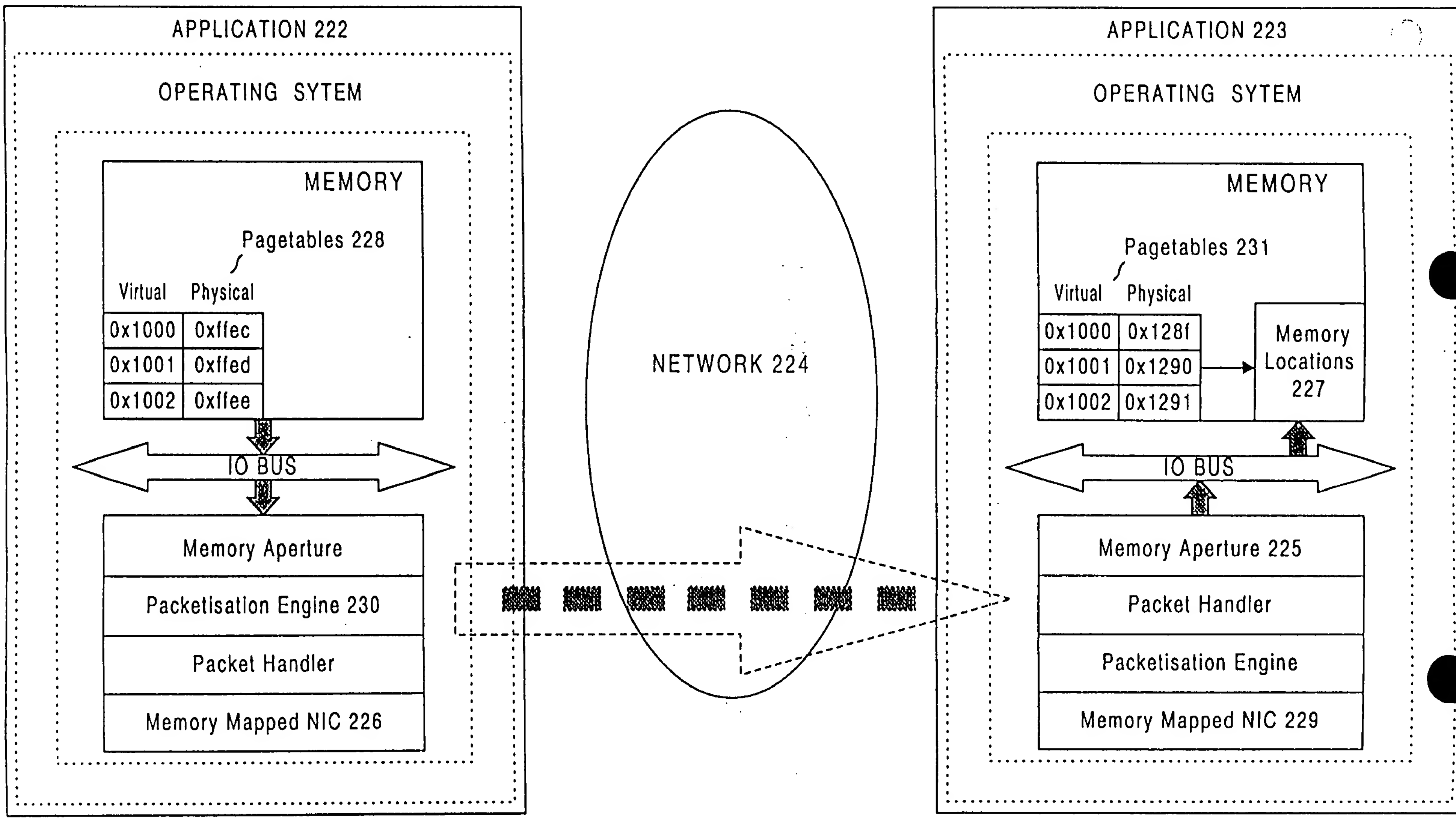


Fig 2. Prior Art. Memory Mapped Networking

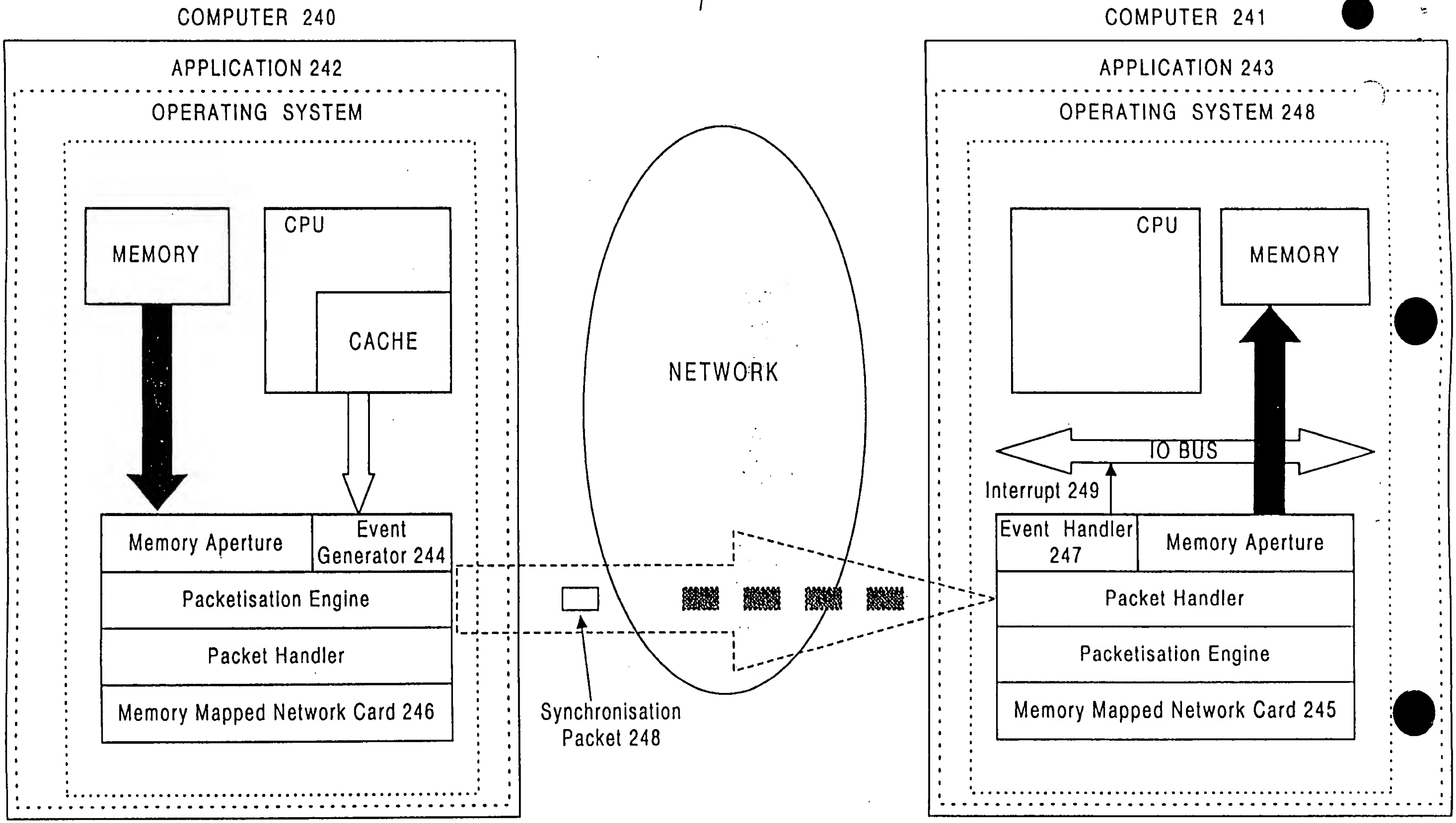


Fig 3. Prior Art. Showing Synchronisation in Memory Mapped Networks

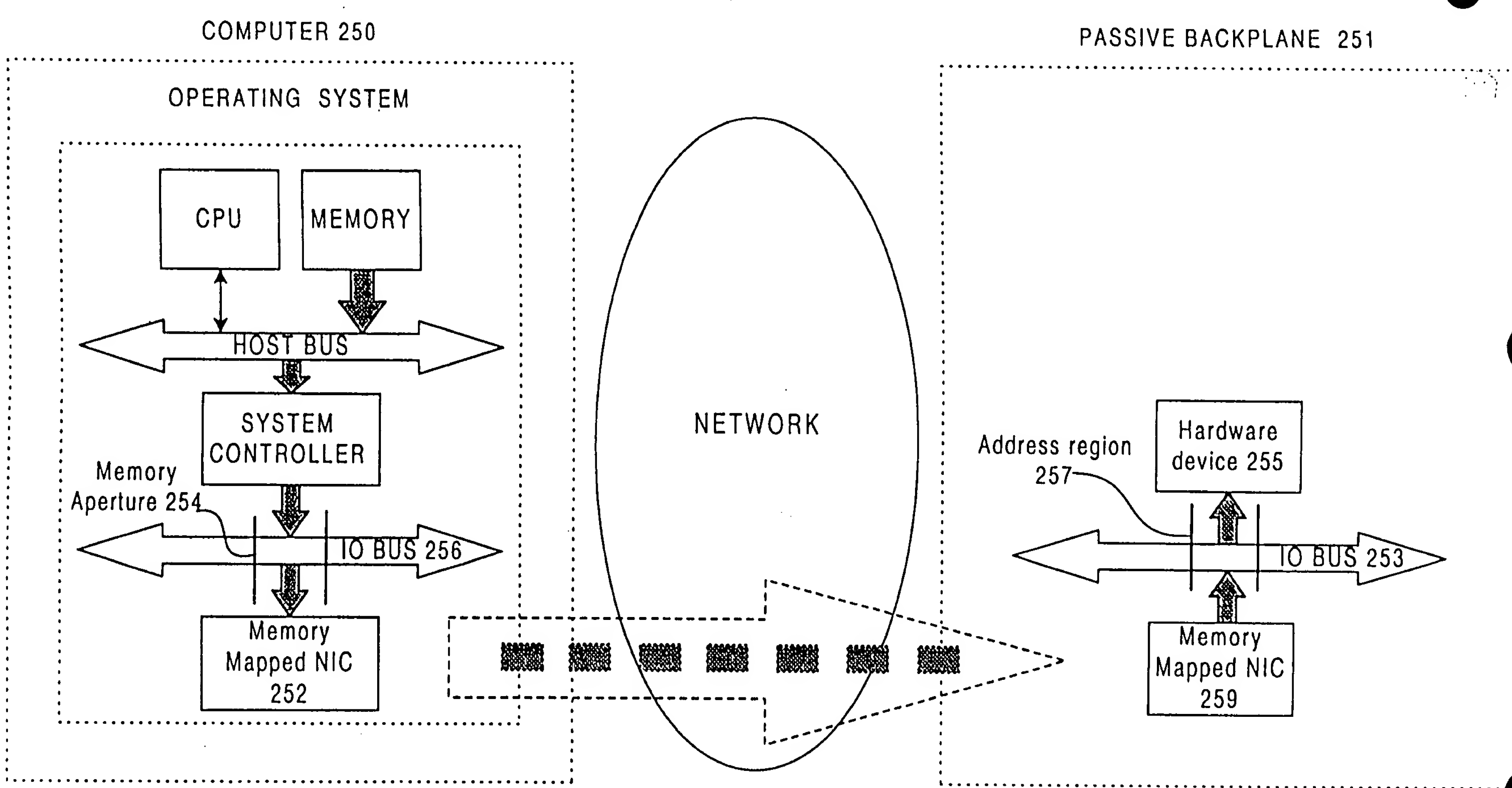


Fig 4. Prior Art. Hardware Communication over a Memory Mapped Networks

This Page Blank (uspto)

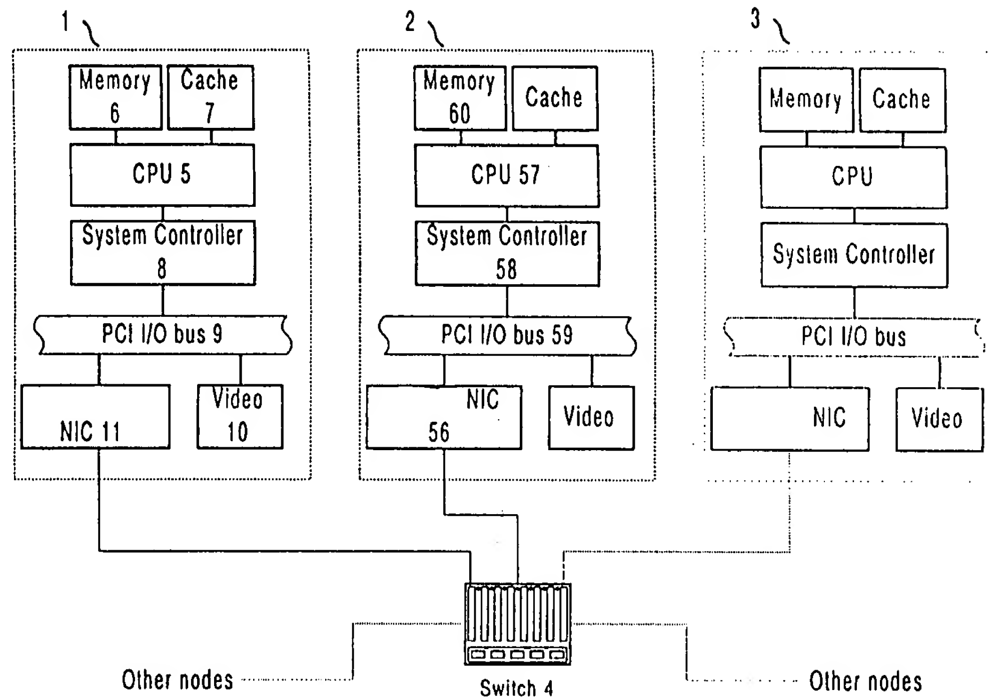


Fig 5. Showing NICs of the present invention being used to construct a Computer Network

6/15

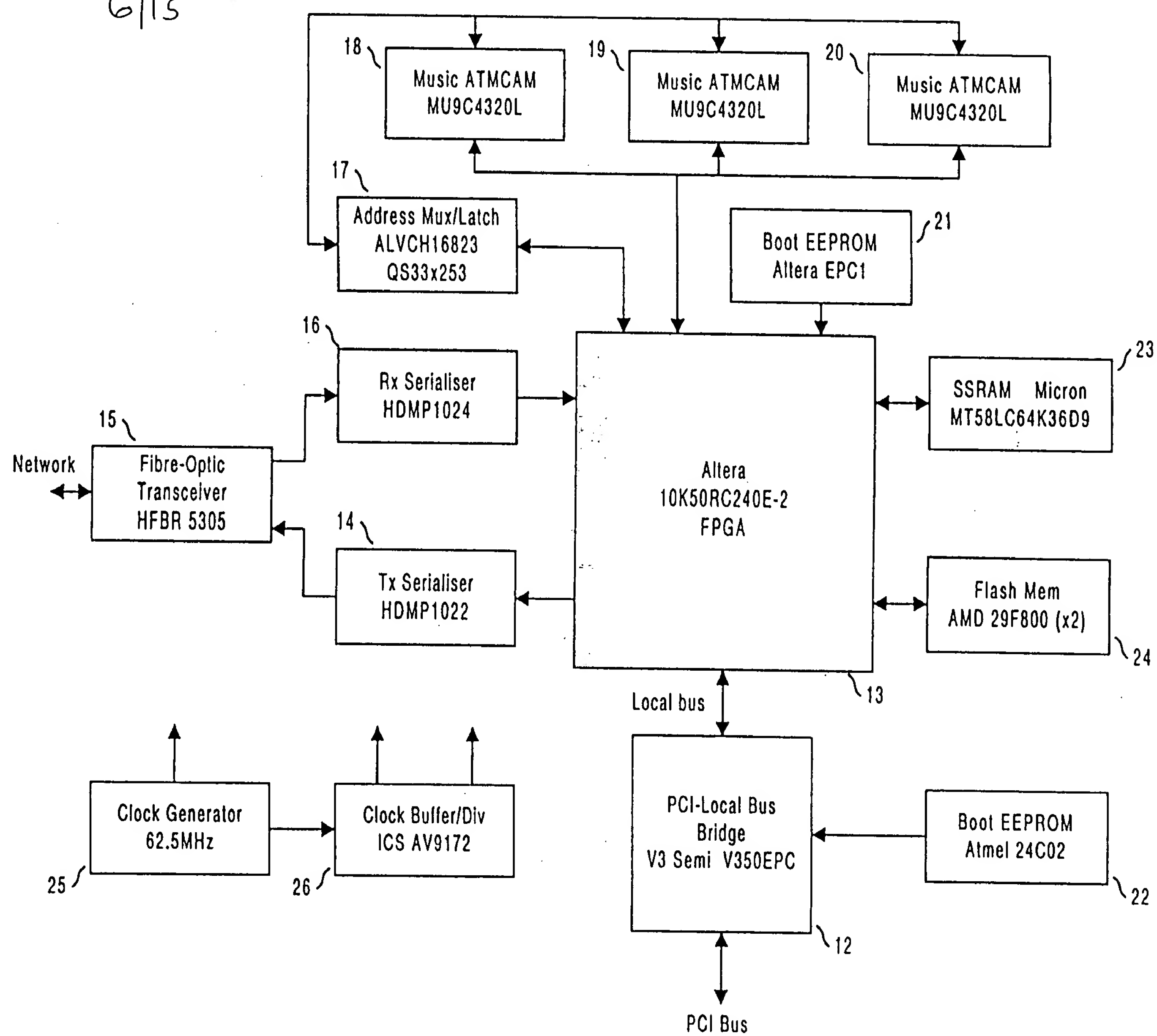


Fig 6. Showing the Functional Blocks of the NIC in one embodiment of the invention.

This Page Blank (uspto)

7|15

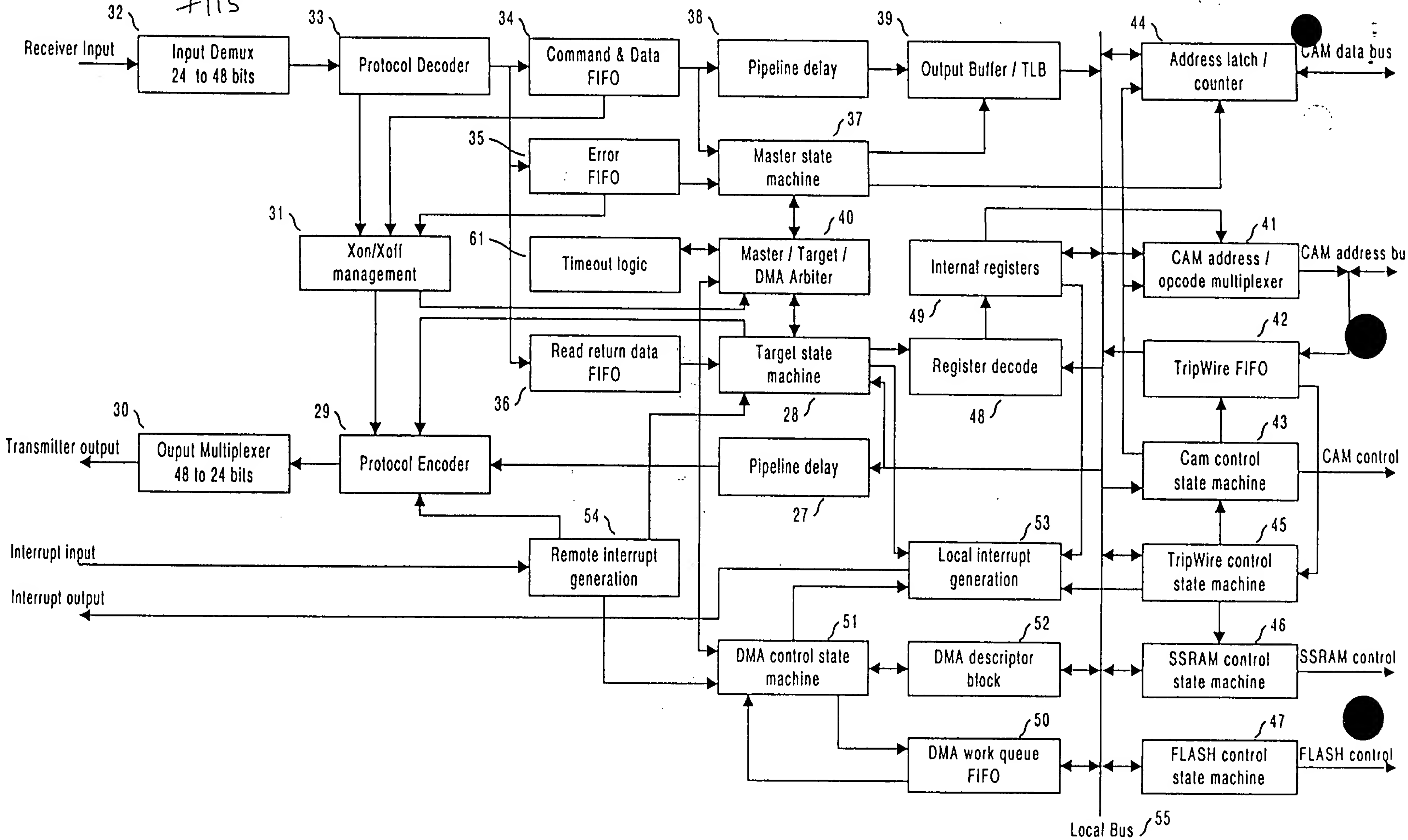
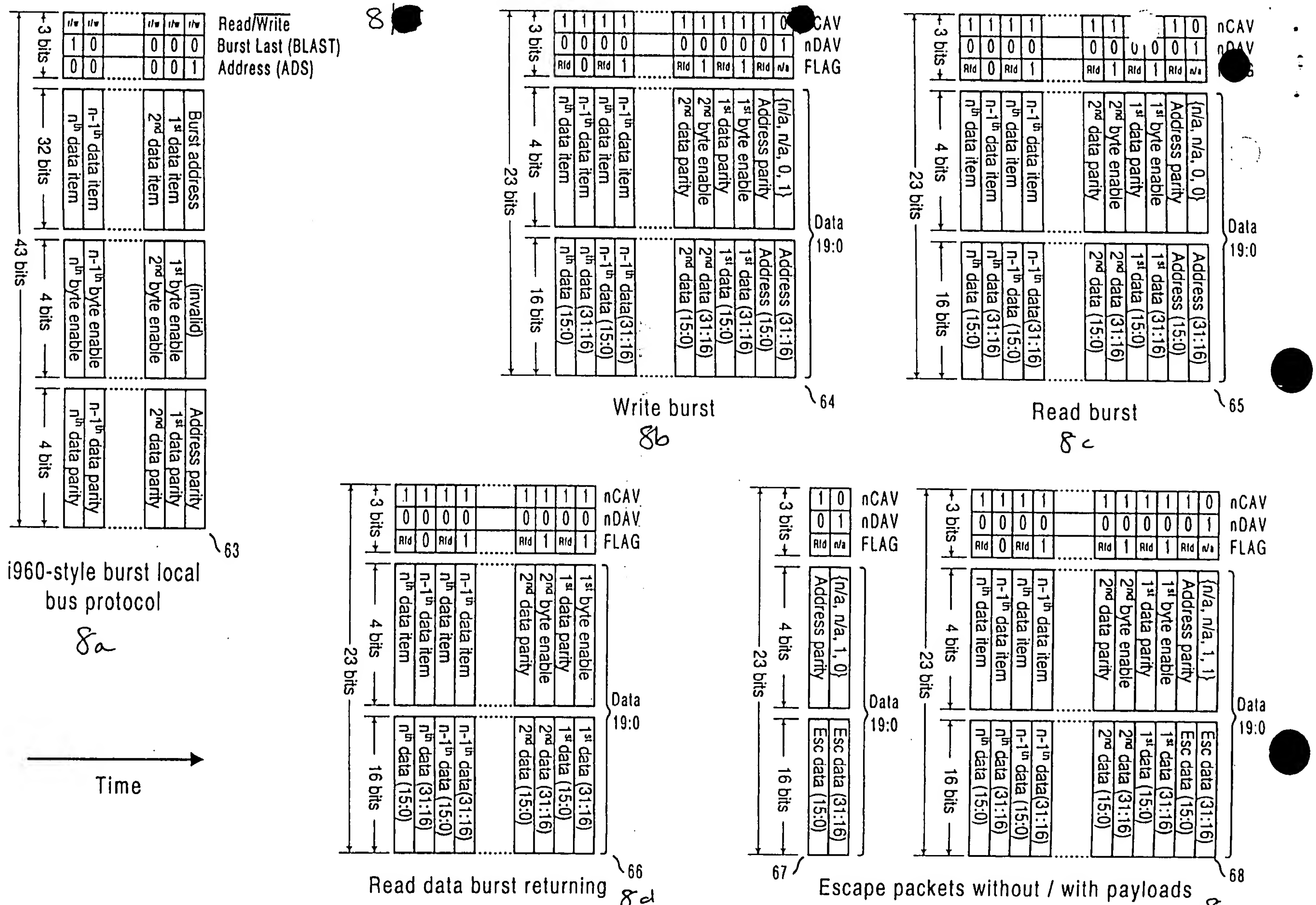


Fig 7. Functional Blocks of the NIC deployed within a Field Programmable Gate Array



Example implementation showing i960-style burst local bus carried on HP-GLINK physical layer

Fig 8. Current protocol embodiment

This Page Blank (uspto)

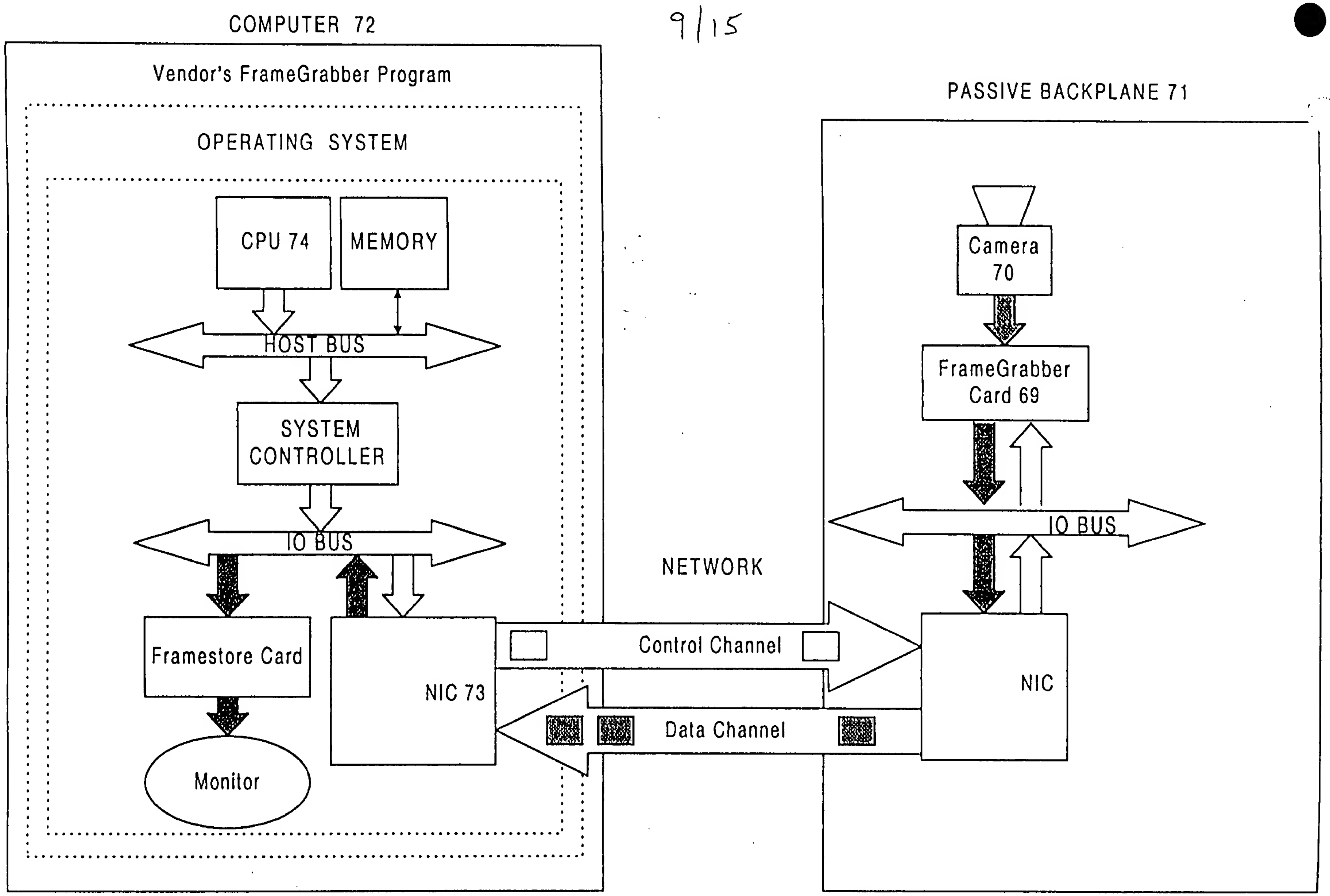


Fig 9. Hardware Communication using the NIC of Fig. 6

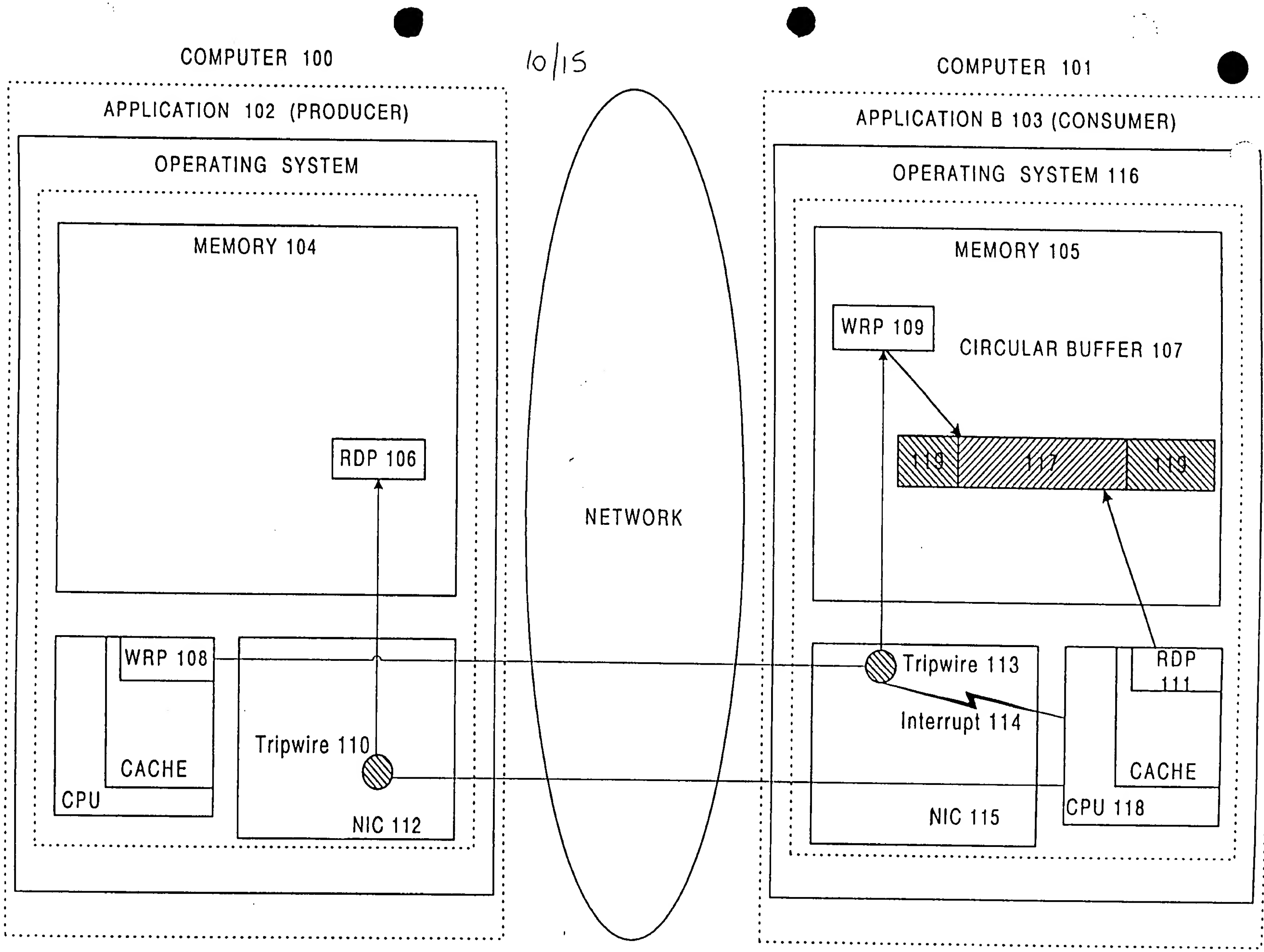


Fig 10. Circular Buffer Support using the NIC of Fig. 6.

11/15

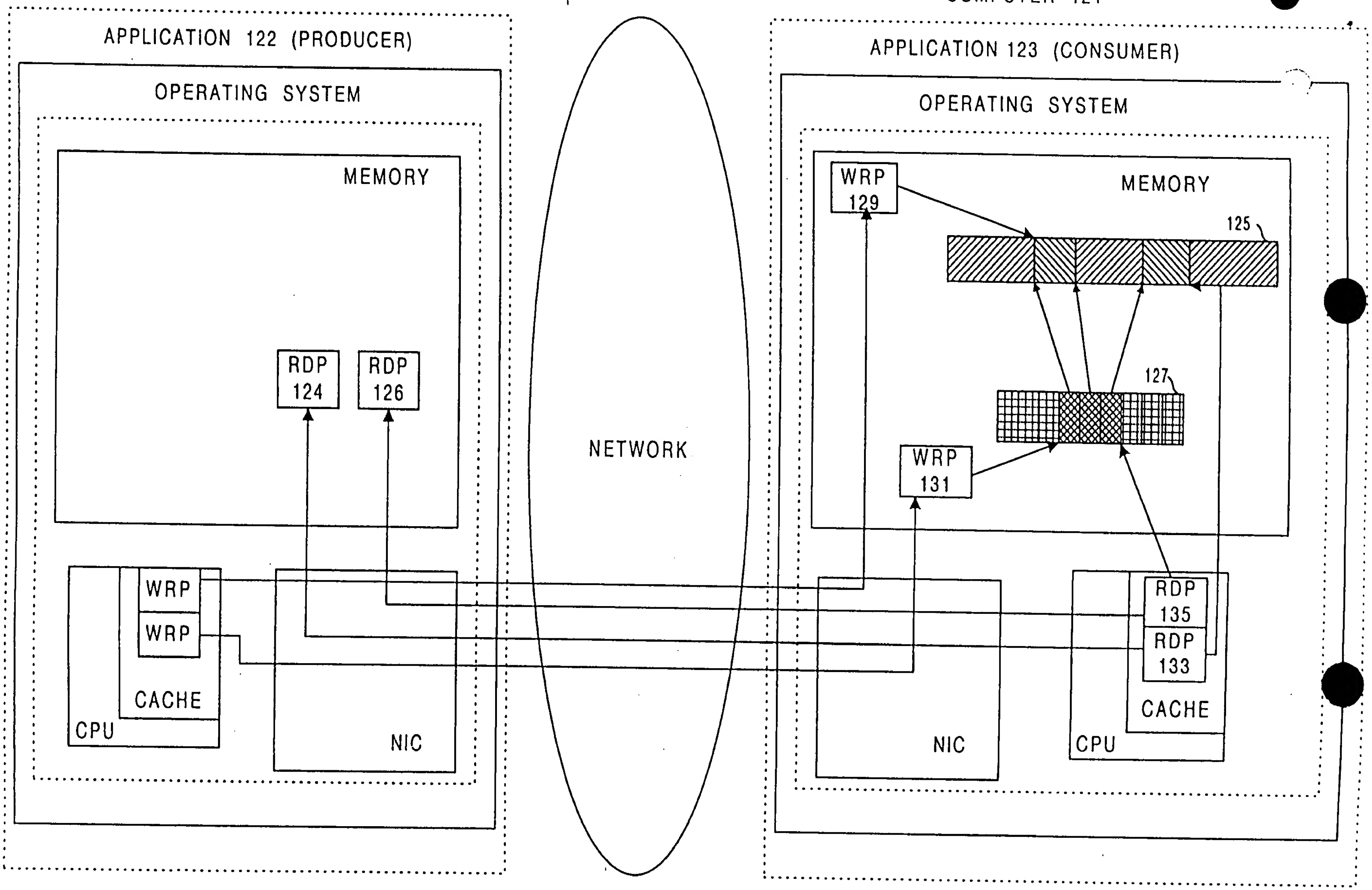


Fig 11. Support for Discrete Message Communication Using Circular Buffers

This Page Blank (uspto)

12|15

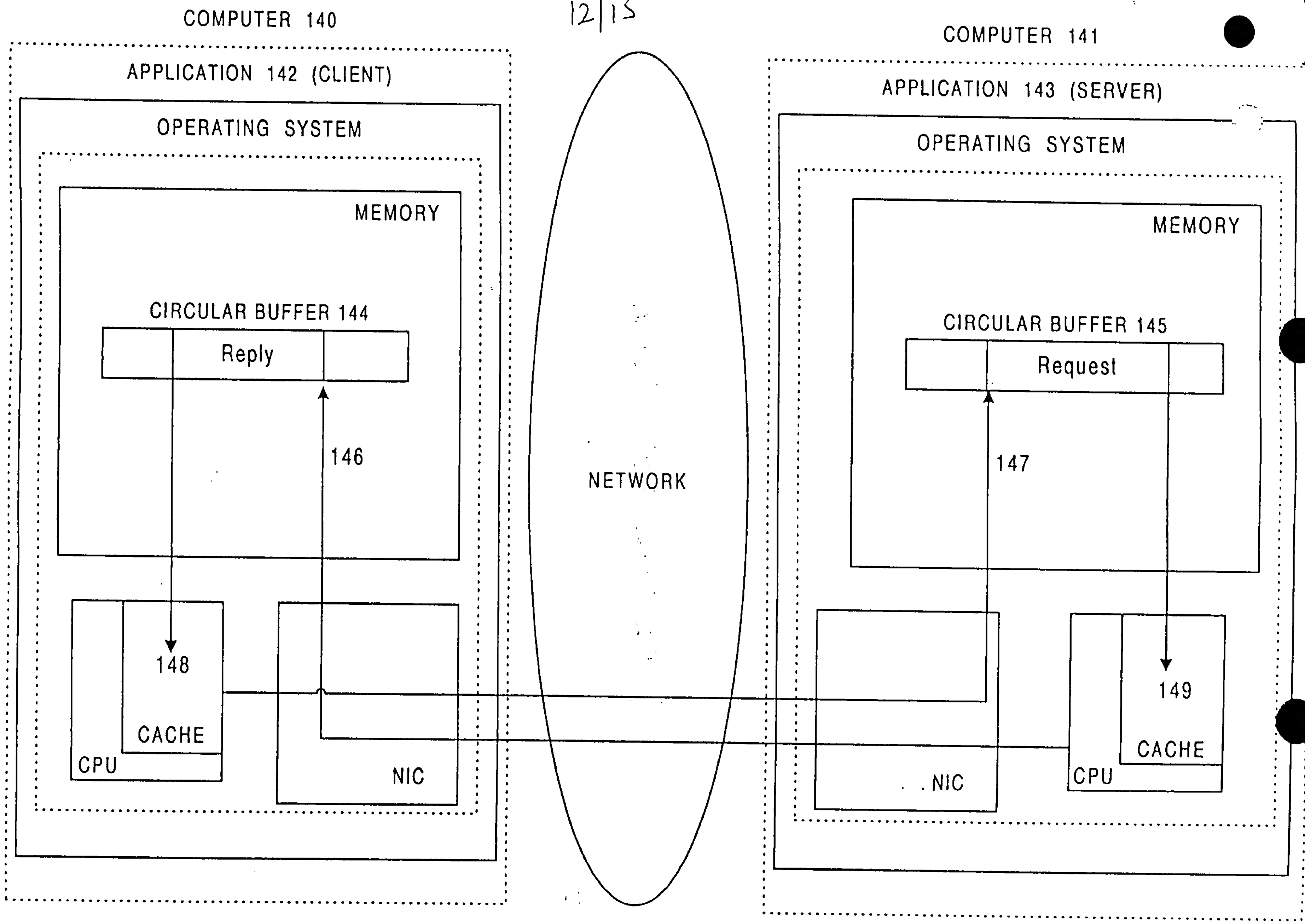


Fig 12. Client-Server Interaction Using two NICs

This Page Blank (uspto)

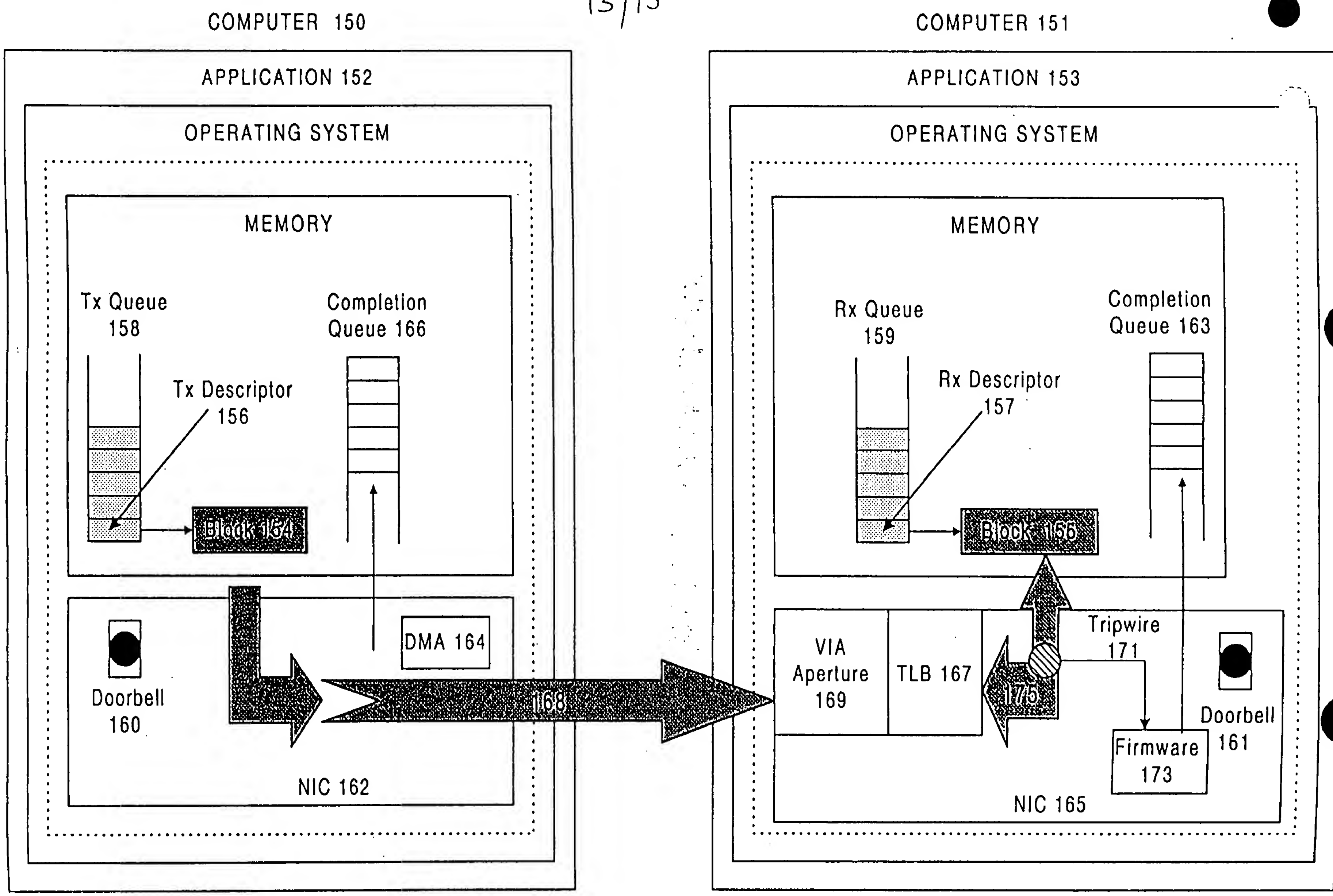


Fig 13. VIA Support

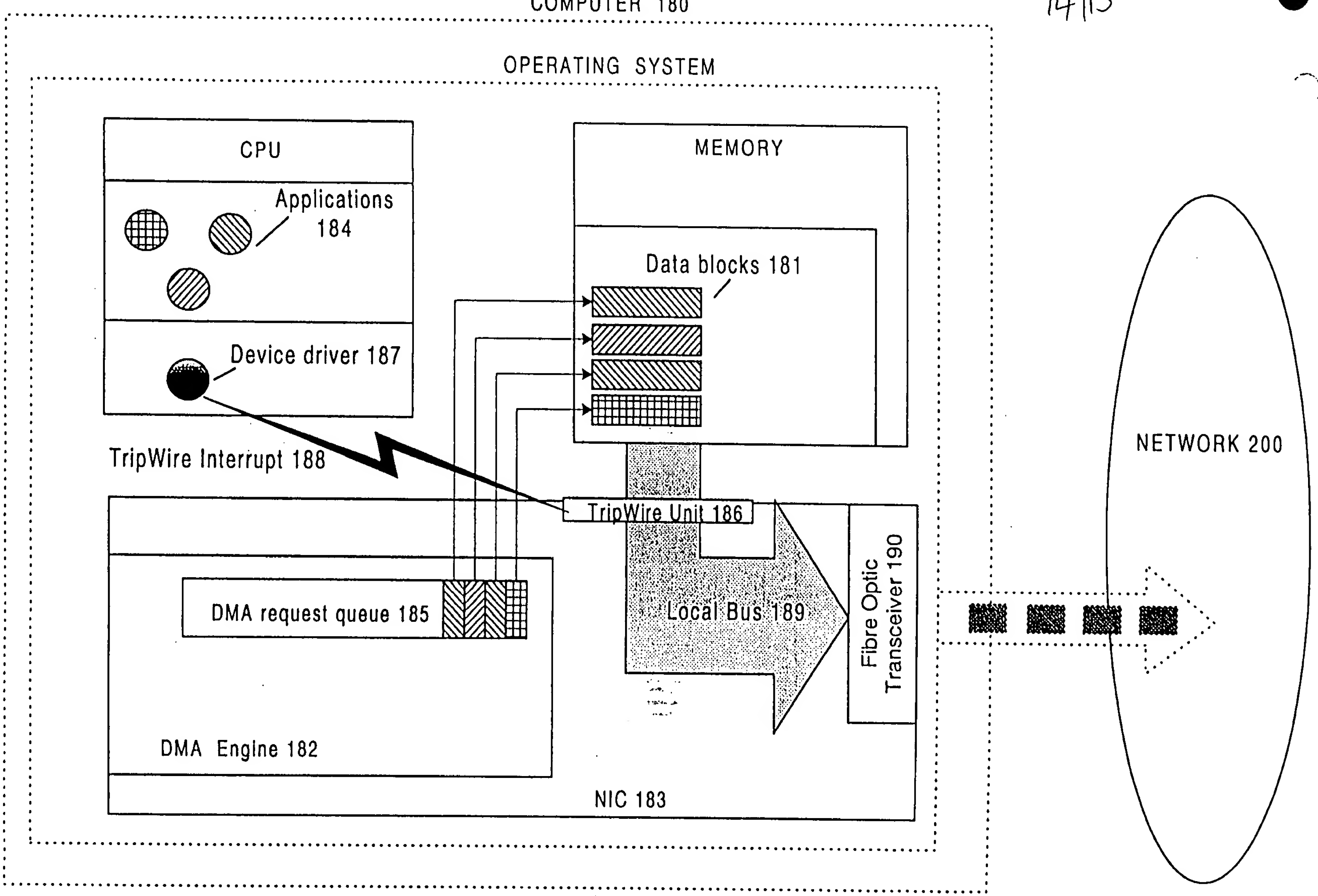


Fig 14. Outgoing Stream Synchronisation

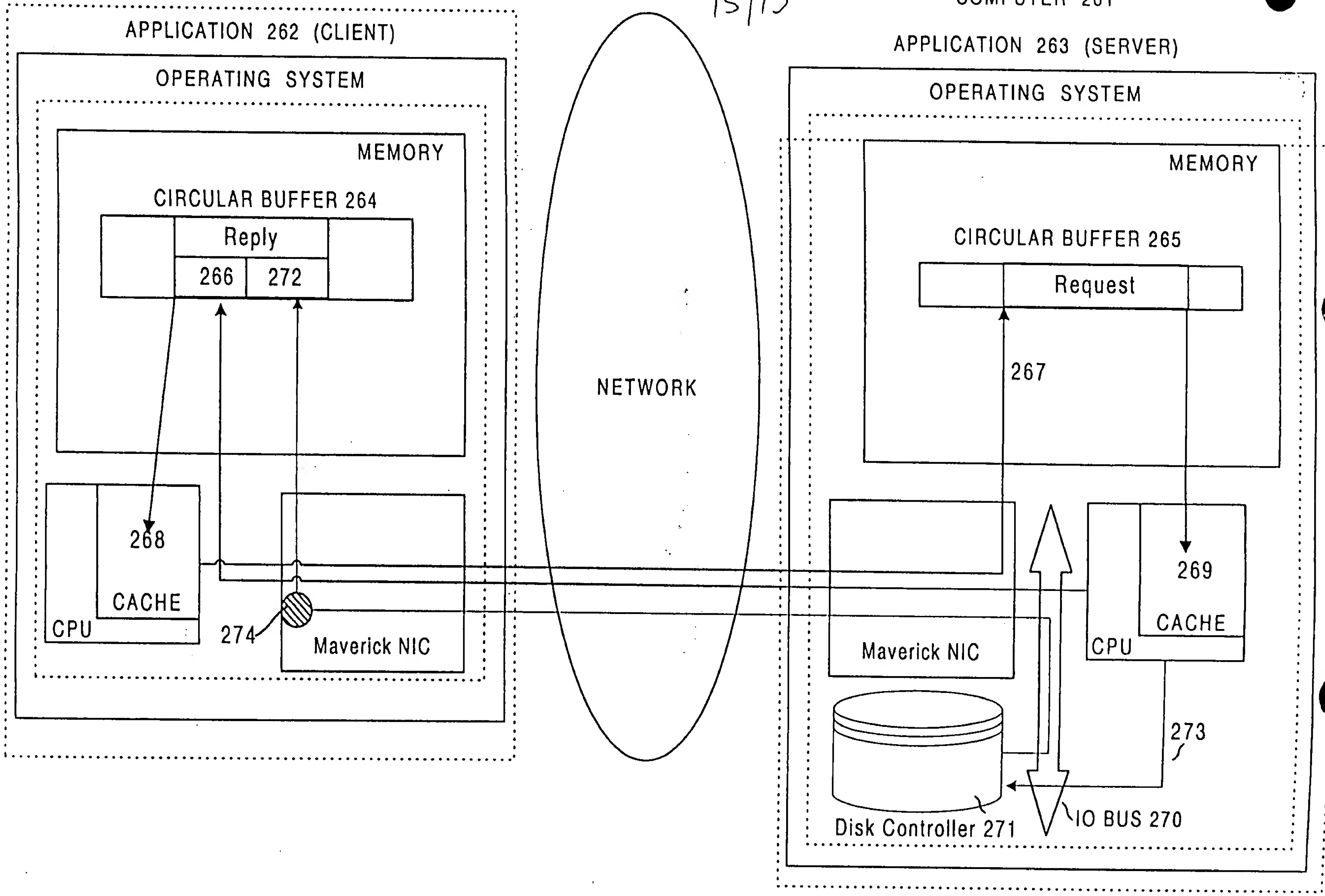


Fig 15. Client-Server Interaction Using a NIC and a Hardware Data Source